Cranfield Institute of Technology

Thabit Sultan Mohammed

# Fault Diagnosis of Distributed Systems: Analysis, Simulation and Performance Measurement

College of Aeronautics

Department of Electronic System Design

PhD Thesis

CRANFIELD INSTITUTE OF TECHNOLOGY

COLLEGE OF AERONAUTICS

DEPARTMENT OF ELECTRONIC SYSTEM DESIGN

PhD Thesis

Academic Year 1991-1992

Thabit Sultan Mohammed

# Fault Diagnosis of Distributed Systems: Analysis, Simulation and Performance Measurement

Supervisor: R. Andrew

# Acknowledgements

I am grateful to my supervisor, Mr R. Andrew, for his guidance, empathy and encouragement.

The help and understanding of the staff of the Department of Electronic System Design are sincerely appreciated.

I am greatly indebted to my family for their moral support, my wife for her encouragement, patience and courage, and my children for their patience and reasonable quietness.

# Abstract

Fault diagnosis forms an essential component in the design of highly reliable distributed computing systems. Early models for diagnosis require a global observer, whereas the diagnosis is shared between the systems nodes in later models. These models are reviewed and their different diagnosability properties reconciled. The design of improved fault diagnosis algorithms for systems without a global observer provides the main motivation for the thesis. The modified algorithm SELF3 [Hoss88] is taken as a starting point.

A number of communication architectures used in distributed systems are reviewed. The properties of diagnosis algorithms depend strongly on the testing graph. A general class of testing graphs, designated as H-graphs, (which are a generalization of $D_{\delta t}$ graphs introduced in [Prep67]), are investigated and their diagnostic properties determined.

A software simulator for distributed systems has been written as the main investigative tool for diagnosis algorithms. The design and structure of the simulator are described. The diagnosis process is measured in terms of diagnostic time and number of messages produced, and the factors upon which these quantities depend are identified. The results of simulation of a number of systems are given under various fault conditions. A modified way of routing diagnosis messages, which, especially in large systems, results in a reduction in both the number of diagnosis messages and the time required to perform diagnosis, is presented. The thesis also contains a number of specific recommendations for improving existing self-diagnosis algorithms.

# Contents

# Figures

# Tables

# Chapter 1

# Introduction

## 1.1 Background

Distributed computing has become an increasingly active area of research and development. The interest has been stimulated by technological changes and by user needs. Advances in microelectronics and the dramatic fall in the cost of both VLSI circuits and networks have changed the price/performance ratio to favour the cost effective design of distributed systems. Moreover, the general increase in the use of computing has led to demands for more sophisticated facilities in terms of speed, reliability, availability, etc.. Such demands are often supported by a general desire to decentralize.

The term *Distributed System* has been used to define a range of systems with centralized systems being at one end and a set of physically distributed autonomous systems at the other . Distributed systems have various applications, and are now considered as a first candidate whenever a new application emerges. These systems are expanding into areas requiring high system availability and where massive parallelism can be exploited to achieve large computational power.

Although most of the design and implementation principles of distributed computing systems have been set, several design issues are still subject to current research. Fault tolerance and reliability are among the issues that are under current investigation. These two issues are steadily gaining in importance as distributed systems become progressively commercialized.

For a number of applications, into which distributed systems are expanding, the implementation of fault tolerance is vital. Such applications include, but are not limited to, safety critical applications, highly available systems, and applications in relatively inaccessible areas. Examples of safety critical applications are transportation (aircraft, trains, air traffic control systems, etc.), industry (control of nuclear power plants, robot

welders, defence systems, etc.), and hospitals (patient monitoring units, life support systems, etc.). Failures in computer systems implemented in these applications , where human lives and expensive equipment are involved, may be catastrophic. In these applications not only must the computation be correct, but any delay associated with fault recovery must be very small. In systems requiring high system availability such as process control in automated factories, control of power generation and distribution, etc., a crash may cause heavy economic losses. The third field of applications of fault tolerant distributed systems is in areas where access to the computer system for the purpose of maintenance is very difficult or even impossible. Examples of these systems are satellites, unmanned spacecraft, underwater stations, etc. . Computer systems for these applications, therefore, need to be capable of maintaining adequate performance until the end of the mission.

Hardware redundancy was the first technique to be implemented to provide fault tolerance, with triplicated computations and voting to detect and correct faults. Among the vast amount of research on the application of fault tolerant distributed systems, two fault tolerant computers for the control of dynamically unstable aircraft have been designed [Hopk78,Wens78]. In both designs, it is required that system failure rate should be less than $10^{-9}$ per hour over the course of 10 hour flight.

Instead of concentrating on hardware redundancy, different techniques for achieving fault tolerance have been investigated in recent research. A main motive behind this research is the assumption that the loss of an individual processor in a distributed system composed of a large number of processing elements may be of little importance in terms of the overall task performed. Thus, it is reasonable to assume that these systems can be designed in a way which allow them to tolerate the loss of one or more processors and still being capable of performing their tasks. It is an essential requirement, however, that faulty processors are not allowed to continue operating in the system without checking, since they might behave in a malicious manner or pass erroneous data to other processors causing a serious system-wide problem. A facility, therefore, has to be implemented to detect and locate faults, and to perform a recovery procedure which allows the system to continue its function.

*Fault Tolerance*, which refers to the ability of computers to withstand failures of some

of their elements, and continue to operate correctly, consists of the following basic steps [Kim79,Rand78]:

(a) Fault detection;

(b) Fault location, which involves performing a set of diagnostic tests;

(c) Analysis of test results to identify faulty elements;

(d) Repair and/or system reconfiguration.

The technique which deals with the interpretation of test results for the purpose of locating faults is *Fault Diagnosis.* This technique is an important tool in the maintenance strategy of computer systems, and in order to minimize the time required in its utilization, it is frequently implemented in hardware.

The theory of fault diagnosis in distributed systems or system level diagnosis (as it is usually called) has received considerable attention over the years. The fundamental model in this area was introduced by Preparata, Metze and Chien [Prep67]. In this model, the system is assumed to be partitioned into units, each of which can perform tests on a subset of remaining units. It is assumed that tests are always accurate when the testing unit is fault-free and the result is two-valued with fault-free (0) and faulty (1) being the only permitted outcomes. Their model is called the PMC model, and diagnosability is based upon syndromes of tests represented on a diagnostic graph G(V,E) in which $V = \{v_i\}$ is the set of vertices and $E = \{e_{ij}\}$ is the set of links over which tests are conducted. The system facility that is involved in gathering test results and is actually performing the diagnosis is excluded from the system model. This facility is referred to as a *global observer*, and is not subject to faults.

Later research has concentrated on more elaborate and more general models, where additional extensions and modifications were found necessary to make the PMC model applicable to actual systems [Frie80]. Some of these researches have recognized that even though the use of a global observer is adequate for many system diagnosis problems, its existence contradicts the principle of distributed systems, since it is unrealistic to assume that such an element is capable of observing all test results without being itself subject to faults. Diagnosis in systems without a global observer has to be performed by the computational nodes.

In distributed systems, the availability of redundant hardware offers a convenient environment for implementing fault diagnosis, since it can be distributed among the processors as part of their computational load. This approach has been referred to as *distributed fault tolerance* [Kuhl80c,Kuhl81,Hoss84]. There are several problems associated with the testing and diagnosis of these systems, which are challenging and important. These problems need to be solved before the systems can be used with confidence especially in safety critical applications. Solutions to these problems have tended to emphasize a system level rather than a logic circuit level approach due to the potentially large number of interconnected units in the system. Research in this area has resulted in a considerably large body of theoretical results, which have yet to be exploited and applied in real systems.

## 1.2 Introduction to Fault Diagnosis

Due to its simplicity, the PMC model proposed by Preparata et al [Prep67] will be used to introduce the theory of system level fault diagnosis. The basic assumptions of this model mentioned in the last section allows for a diagnostic system to be modelled as a directed graph. In this graph a directed edge $e_{ij}$ represents a test in which node $v_i$ tests node $v_j$. It was further assumed that while the system is performing its diagnosis, the status of its units as faulty and fault-free remains unchanged (i.e., the model considers permanent faults only). Practical application of tests is assumed to be carried out by applying a controlled stimuli and observing the responses. There might be other methods such as; comparison or evaluating the output of a self-checking circuit. In any case, however, the test is assumed to be complete, and the diagnosis of the system is evaluated by decoding the *syndrome*, which is a vector formed by collecting the weights of test links participating in the testing process. Each different syndrome is associated with a different fault pattern, hence it will be impossible to differentiate between two faults if the syndromes associated with them are the same. Moreover, test results of some units can be affected by the presence of other faults in the system. This problem is referred to as *test invalidation.* In the PMC model the type of test invalidation assumed has been called *symmetric invalidation*, Figure 1.1.a, where it was assumed that if the tester unit is fault-free then it will correctly determines the status of the units it tests. However, results of tests performed by a faulty tester are unspecified, and hence unreliable.

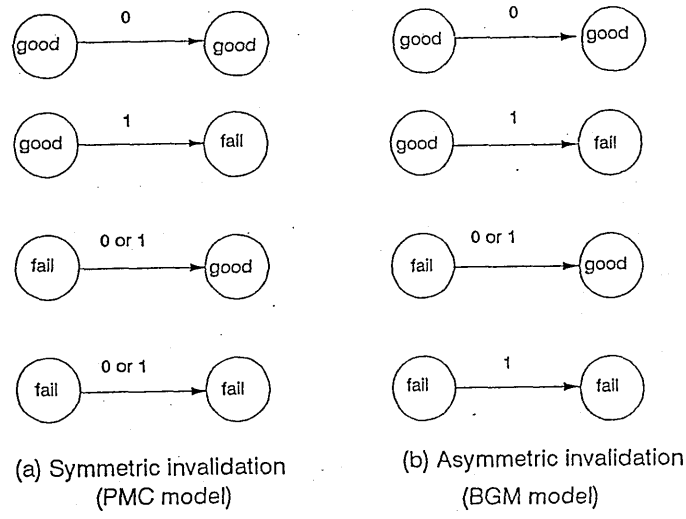



Figure 1.1: Symmetric and asymmetric test invalidations

*Definition 1.1:* A system is referred to as *self diagnosable* if it has the ability of clearly identifying its faulty subsystems up to a given bound.

Based on the PMC model, two measures of system diagnosability were defined;

*Definition 1.2* : A system is said to be *one step t-fault diagnosable* (or *t-fault diagnosable without repair*) if a single application of tests allows for the identification of all faults, provided the number of faulty units in the system does not exceed $t$.

*Definition 1.3 :* A system is said to be *sequentially t-fault diagnosable* (or *t-fault diagnosable with repair*) if the application of tests allows for at least one faulty unit to be identified, provided the number of faulty units does not exceed $t$.

The main results of Preparata's et al work are explained in the following theorems:

*Theorem 1.1* : A system S is one-step t-fault diagnosable, only if the number of its units $n$ is greater or equal to $2t + 1$, and each unit is tested by at least $t$ other units.

*Theorem 1.2* : If $n \geq 2t + 1$, it is always possible to provide a connection to form a one step t-fault diagnosable system.

*Theorem 1.3* : A system S is an optimal (i.e., having the minimal number of test links) one step t-fault diagnosable system if $n = 2t + 1$ and each unit is tested by exactly $t$ other units.

*Theorem 1.4* : There exist a class of systems with a number of test links $L = n + 2t - 2$, that are sequentially t-fault diagnosable.

The conditions of Theorem 1.1 are necessary but are not sufficient as will be shown in the following example.

Example 1.1: Consider the system shown in Figure 1.2, which consists of four units and each unit is being tested by at least one other unit. The single fault test syndromes of this system are shown in Table 1.1. In this Table, (x = 0 or 1) refers to an unreliable test result, and since the test syndromes for a single fault in units 1 and 2 may be identical, the system is not one-step one-fault diagnosable although it does satisfy the conditions of Theorem 1.1 for $t = 1$.


Figure 1.2: A system which satisfies the conditions of Theorem 1.1 for $t = 1$.

| | Syndromes | | | | |
|---|---|---|---|---|---|
| Faulty nodes | 12 | 21 | 23 | 34 | 43 |
| 1 | x | 1 | 0 | 0 | 0 |
| 2 | 1 | x | x | 0 | 0 |
| 3 | 0 | 0 | 1 | x | 1 |
| 4 | 0 | 0 | 0 | 1 | x |

Table 1.1: Single fault test syndromes for the system of Figure 1.2.

The basic motivation behind the investigation of sequential diagnosability is to reduce the number of test links. It can be noticed from *Theorem 1.4* that $L = n + 2t - 2$ test

links are required for a sequentially t-fault diagnosable system, whereas an optimal one-step t-fault diagnosable system requires $nt$ test links. The implementation of sequential diagnosability involves a multi-step diagnostic procedure of repetitive identification and replacement of faulty units by fault-free ones until all faulty units are replaced. The replacement process is practically an off-line repair, which is not within the scope of this thesis, hence when the term "t-diagnosable" is used it will mean "t-diagnosable without repair" unless stated otherwise.

In another diagnosis model proposed by Barsi, Grandoni, and Maestrini [Bars76], known as the BGM model, a different type of test invalidation called the *asymmetric invalidation* was proposed, where it was assumed that tests performed by fault-free units always give correct results as pass or fail, while tests performed by a faulty unit on another faulty unit must always fail. Symmetric and asymmetric invalidations are shown in Figure 1.1. The difference between the two types is that under symmetric invalidation only test results performed by fault-free units are guaranteed, while under asymmetric invalidation a test on a faulty unit is guaranteed to fail regardless of the condition of the tester.

Designing a system under the assumption of asymmetric invalidation of tests seems to be of better value in terms of higher diagnosability and by offering an easier way to formulate diagnosis algorithms. The introduction of the asymmetric invalidation was justified by the assumption that a tester is a unit with a large computational capability,therefore when it conducts a test, a significant amount of stimuli will be applied. It may be reasonable to assume that if the unit under test is faulty then it will react to the stimuli with a different response from the expected, even if the tester unit is faulty. This makes it clear that the nature of the test and the way test results are evaluated are two important factors that need to be considered when a type of test invalidation is to be chosen.

The diagnosis algorithm implemented in this thesis considers symmetric invalidation.

## 1.3 Literature Survey

Even though the PMC model is not applicable to actual systems, it has laid the groundwork for later research that has attempted to generalize the model and add more realistic constraints associated with actual systems. Most of the work on system level fault diagnosis has focused on generalization of: the system diagnostic graph, the possible test results, and diagnosability measures.

Russell and Kime [Russ75a,Russ75b] proposed a generalization of both the PMC and the BGM models. Their model, which is known as the RK model, presents a general test invalidation assumption. It was assumed that if a unit is tested by the cooperation of several other units, then if any one of these units is faulty the test result will be invalid. This assumption was called *multiple invalidation per test* [Kreu87]. The RK model may be formulated either with symmetric or asymmetric invalidation [Holt81].

In the PMC model it was assumed that all faults are equiprobable. In a generalized model, however, Maheshwari and Hakimi [Mahe76] took into account the probabilistic nature of fault occurrence, and hence a probability of failure, as a positive valued weight, has been associated with each unit in the system. By considering this, several fault patterns may be generated with different probabilities, the objective is then to identify the most probable fault pattern. A diagnosis algorithm of complexity $O(n^3)$ for this model has been proposed in [Dahb86].

In [Prep67] the two proposed measures of system level diagnosis, the one-step t-fault diagnosability and sequential t-fault diagnosability, allows only faulty units to be replaced. Thus, for a system, which is sequentially t-fault diagnosable, tests must be repeated t-times(steps) before all faulty units in the system are identified and replaced. Friedman [Frie75] has defined another measure known as *m-step t/s diagnosability*, where by $m$ applications of diagnostic tests; a set of $t$ faulty units can be diagnosed and repaired by replacing at most $s$ units. This diagnosability measure has been considered for regular systems, and more work has been done on the special case when $s = t$ [Karu79,Yang86], where far fewer total tests are required for diagnosis.

Among the many algorithms that have been developed to solve the diagnosability problem of a t-fault diagnosable system, Dahbura and Masson [Dahb84] presented an $O(n^{2.5})$ diagnosis algorithm, which was claimed to be the most efficient for a general case of t-fault diagnosable systems. Algorithms for special cases of these systems were presented by Meyer and Masson [Meye78] and Hakimi and Chwa [Haki81].

Generalizations of the models proposed in [Frie75,Karu76,Mahe76,Prep67] were studied by Sullivan [Sull86]. A number of polynomial time algorithms to solve the t-diagnosability problem for the symmetric invalidation model were presented for the first time in this work.

In practice, most of the faults are either transient or intermittent. In contrast to the permanent fault diagnosable system, where test results of fault free units are always evaluated correctly. Mellela and Masson [Mall78] assumed that a fault-free unit might be judged as faulty, while it is in fact intermittently faulty but the test applied is insufficient to realize this. Thus for intermittent fault diagnosability the test routine need to be applied repeatedly before a faulty unit is diagnosed.

Kuhl [Kuhl80c] proposed that for a large system, where units are complex and their tests might be fairly complex and time consuming, the syndrome produced by applying a series of tests according to the PMC model will require a relatively long time, and it is possible that some units become faulty during the testing period. The PMC model was modified, where each node in the diagnostic graph is labeled with the time(s) at which the corresponding tests are performed.

In [Rang88] Rangaranjan et al proposed a diagnosis algorithm for locating faulty and fault-free units in systems comprising a number of processors that are being allocated similar computational tasks. The algorithm is based on a comparison approach. According to this approach, which was proposed in some earlier work [Haki81,Maen81], outputs of tasks, whose execution is completed by the processors, are compared among themselves. Each processor which completes the execution of the task compares its results with other processors, and by analyzing the comparison it should be able to decide upon its own status, as faulty or fault-free. The possibility of a faulty processor considering itself as fault-free or vice versa is guarded against by the use of a self-checking diagnostic hardware, which is assumed to be robust. The comparison process can be considered as an additional task for the processor, which requires a constant time, and being executed during system operation. The algorithm may be suitable for a class of systems composed of homogenous multiprocessors, designed to exploit parallelism of tasks. Examples of which are systems performing matrix operations or image processing, where a single instruction is given to all the processors to perform similar operations on many elements of the matrix or the image.

An alternative to the implementation of the "global observer", which is the unit responsible for system diagnosis in the PMC model and its modification, the concept of *roving diagnosis* was proposed by Nair [Nair78]. This approach is based on a roving graph,

which is a subgraph of the system diagnostic graph, representing a time-varying part of the system. An important issue on which this approach heavily depends is to ensure that the first diagnosis identifies some fault-free units. These units will then be used to diagnose another part of the system, and hence each part diagnose a second part, while the remainder of the system continues normal operation. Thus, throughout the diagnosis process, there will be a subsystem of diagnosing and diagnosed units, which roves through the system until all its parts are diagnosed. In this approach only good units are allowed to perform tests on other units, therefore the problem of test invalidation is avoided. The system is required to be designed so that good initial nodes, to be responsible for initiating the diagnosis, can always be identified, however, methods for achieving this requirement were not considered in detail. Because only part of the system is involved in diagnosis at any time, the system will have high availability.

In all models that are based on one-step diagnosis a set of tests are selected and scheduled first, then system diagnosis is evaluated from analyzing results of these tests. As an alternative, Nakajima [Naka81] has proposed the concept of *adaptive diagnosis*, where tests can be chosen adaptively so that a fault-free unit can be diagnosed. This unit is then used as a tester to identify all faulty units in the system. It was assumed that every unit is capable of testing every other unit. The idea was further explored by Hakimi and Nakajima [Haki84], where an adaptive diagnosis algorithm was proposed. Tests are selected and performed one at a time with the next test to be applied being a function of previous test results. The advantage of adaptive diagnosis over one-step diagnosis is that the number of tests required for performing adaptive diagnosis is considerably fewer. On the other hand, longer time is normally required for performing adaptive diagnosis, hence it is not recommended when there is high probability of units becoming faulty during the testing period.

Problems in the area of system level diagnosis of distributed systems have been simplified by assumptions, which help in producing mathematical models, but they might not be closely adhered to the requirements of designing fault tolerant systems. From the practical standpoint, these assumptions may be considered unrealistic [Dahb87]. Centralized diagnosis (i.e., diagnosis with global observer) is an example of these assumptions. In order to avoid the aforementioned implications associated with the centralized diagnosis, Kuhl and Reddy [Kuhl80a,Kuhl80b,Kuhl80c,Kuhl81] suggested the distributed diagnosis,

where diagnosis tasks are distributed among the system units themselves, and instead of depending on the global observer for performing the diagnosis, each fault-free unit in the system arrives at its own diagnosis by testing its neighbours and receiving the results of tests that they have performed or obtained from other units. This approach was modified to include link failures. One of the refinements of this technique will be considered in this thesis, therefore the original work and its modifications will be investigated and explored in chapter (3), which will be devoted for this purpose.

## 1.4 A Project Overview

The work in this thesis has been directed towards simulating fault diagnosis of distributed systems by developing a software tool, which is useful for the analysis of system diagnosis and to provide a clearer view of the behaviour of a distributed system, which implements a diagnosis algorithm. The work can be considered as a step towards reducing the gap between theoretical results and their applications.

In the simulator, a distributed system is modelled into a set of nodes interconnected by a set of communication links. The set of nodes represents the processing elements, while the set of communication links represents a subset of the system interconnection network. Using these links, tests are conducted among the processors and diagnostic information are routed in the form of messages. Passing messages is the only way by which the processing elements of the system can communicate.

The system specification is presented to the simulator in a tabular form, which represents the adjacency list of the graph G(V,E). This type of data structure is favourable for representing graphs [Gibb85]. The attributes of each node, that are used for the purpose of diagnosis by the node, fault specification of the system, and pre-simulation initializations are formed in a high level language.

Processors in a real system execute messages according to the order of their receipt. In order to perform a correct simulation, the simulator is required, for every node, to process the messages in the same order as the equivalent processor in the real system. A simulation algorithm has been implemented, and one of its responsibilities is to make sure that the necessary order of execution is being followed. As for diagnosis, a distributed algorithm, which was proposed by Hosseini et al [Hoss88] is considered as the starting point for the work. Many assumptions and modifications were found necessary for adopting

the algorithm to the distributed system environment. Defining new types of messages for the actions that were not specified in the algorithm by a message form, implementing a unified format for all messages, and appending a time stamp to every message for the purpose of ordering, are some of these assumptions or modifications.

Fault tolerance and fault diagnosis capabilities of a distributed system are related to its topology [Prad82], where the topology of a system defines the interconnection architecture of its processors. The complexity of the routing algorithm, and the message delay are both related to the regularity of the system architecture. A highly regular system allows for simple routing and shorter message delay than irregular structure. The hypercube is a well-known example of a regular topology.

Graphs are normally used to represent system architectures. A class of highly regular graphs, which are easy to construct, has been proposed and applied to the simulator. The hypercube and the proposed graphs are used to study the performance of the diagnosis algorithm. The implications that have been noticed during this study were analysed and improvements to the performance of the algorithm were implemented.

# Chapter 2

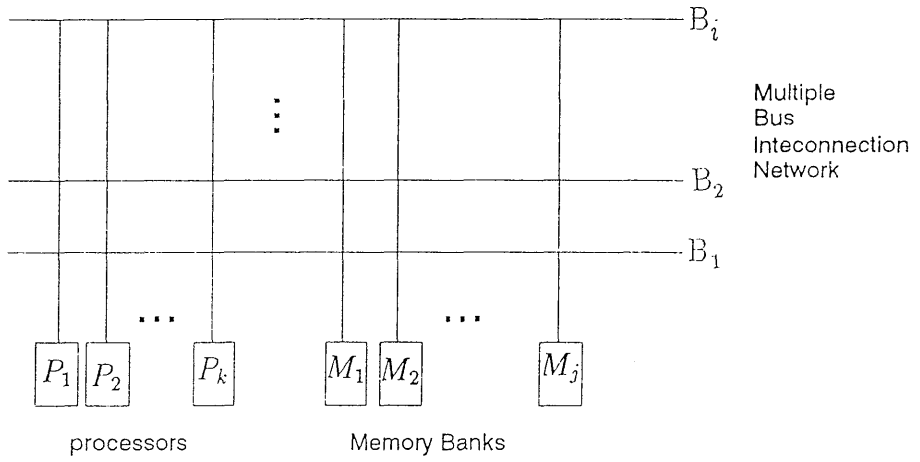
# Communication Architectures for Distributed Computing Systems and Fault Diagnosis

## 2.1 Introduction

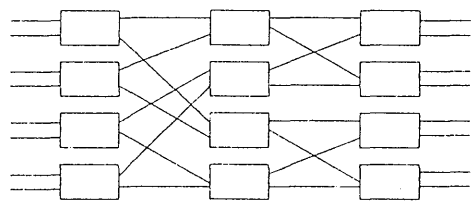
Distributed computing systems have been categorized into multiprocessors and multicomputers according to their internal structure. In the former, there exists a main memory that is being shared between processors and can be directly accessed by each one of them. In the latter, each processor has its local memory, which can only be accessed by the processor itself. A simple overview of the structure of both categories is of a computer system being composed of a set of processors that are physically distributed, and connected to a communication network, which supports the communication between them. Each processor is capable of communicating with all other processors that are connected to the network either directly or indirectly.

Communication networks for computer systems cover a spectrum of architectures , which extend from a network in which few computers are connected over long distances, up to the very large scale integration (VLSI) systems, where a large number of processors are interconnected on a single chip. For a system that is distributed throughout one building, a local area network such as the Ethernet can be used.

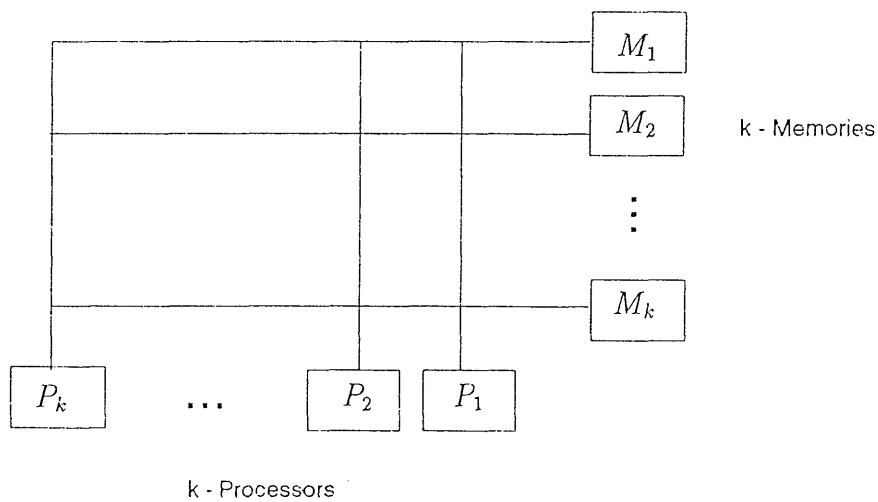
Many of the network architectures used in a multiprocessor system are based on crossbar [Sawc87], multistage networks [Kuma87], and multiple bus networks [Mudg74]. Architectures such as star, ring, tree and hypercube are considered in multicomputer systems. Examples of multiprocessor and multicomputer communication network architectures are shown in Figure 2.1 and Figure 2.2 respectively, and more of these architectures can be found in [Adam87,Davi79,Farb72,Sieg79,Thur74].




(a) Multiple Bus Network



(b) Multistage Network




(c) Crossbar Network

Figure 2.1: Examples of network architectures used in multiprocessor systems

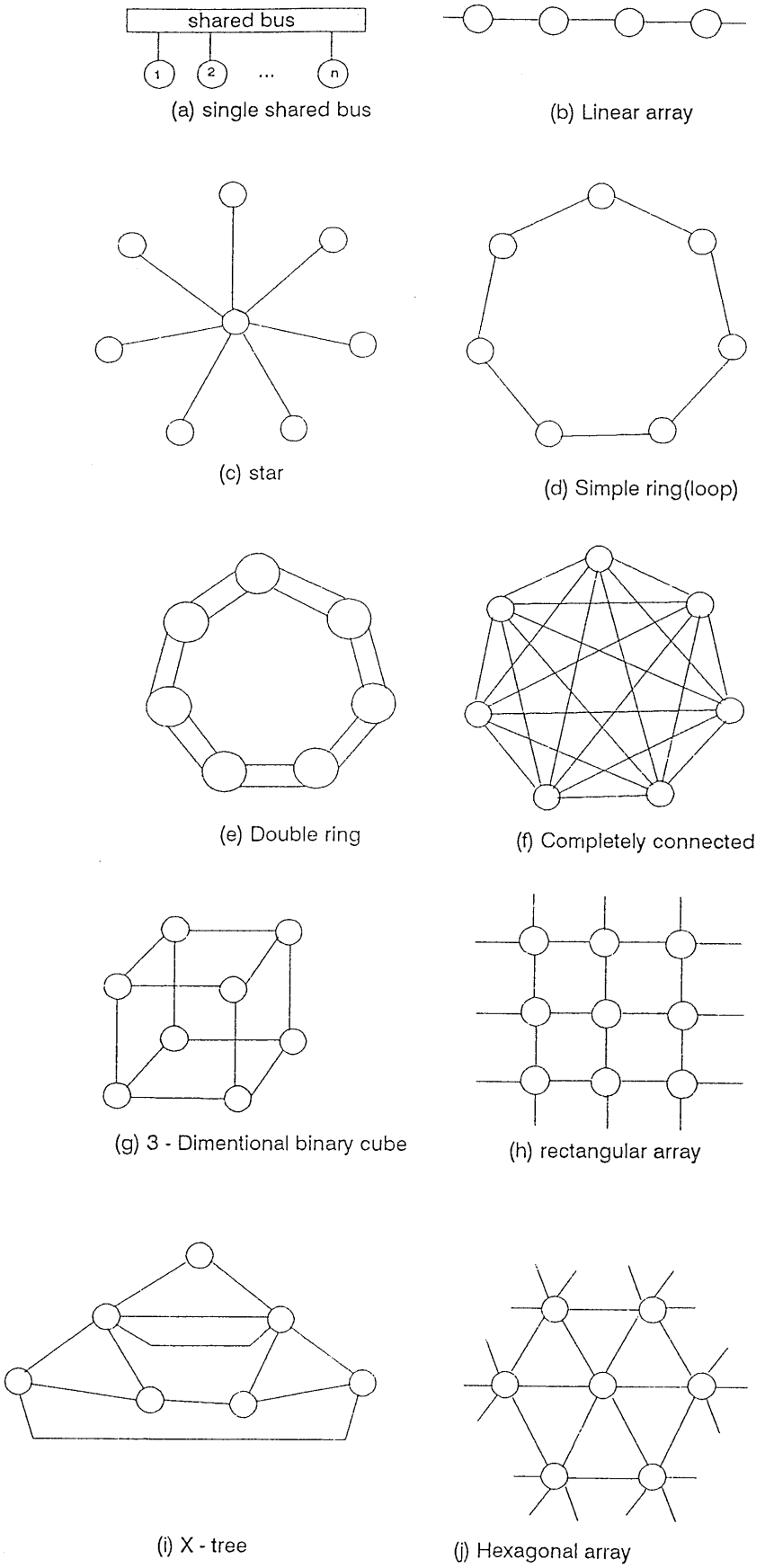



Figure 2.2: Examples of network architectures used in multicomputer systems

The design of a communication network can be featured by the following operational properties:

**Switching methodology** : Circuit switching and packet switching are the two main methods that are used for moving data in communication networks. In circuit switching, the path that is being used by a processor for transmitting data will be occupied for the duration of transfer. In packet switching, however, data is divided into small packets and transmitted through the communication network. A packet might pass through a number of intermediate processors before reaching its final destination in a store-and-forward manner.

**Operation Mode** : This property is related to the timing of communication and it is divided into synchronous and asynchronous. A central clock controls the operation in the synchronous type, while an independent operation of processors without global clock is being implemented in a network using asynchronous mode.

**Control strategy** : Based on this property, communication networks are classified into centralized and distributed. A global controller handles all requests in centralized control, while in distributed control, requests of different processors in the network are handled independently.

Examples of distributed systems which employs synchronous operation mode and centralized control strategy in their communication networks are the SIMD machines [Sieg85]. The acronym SIMD stands for single-instruction stream multiple-data stream. In these systems, a global controller broadcasts instructions to the processors which are connected to the network. All processors will execute the same instruction (single instruction stream) at the same time (synchronously) using data from their own memories (multiple data stream). In a different type of systems referred to as MIMD, however, asynchronous operation mode and distributed control are employed.

In this and next chapters, the distributed system is assumed to be a multicomputer that is interconnected by a communication network characterized as packet-switched with distributed control and asynchronous operation mode. A unit in these systems is assumed to consist of a processor with a local memory and attached to the network through a switching element, Figure 2.3.

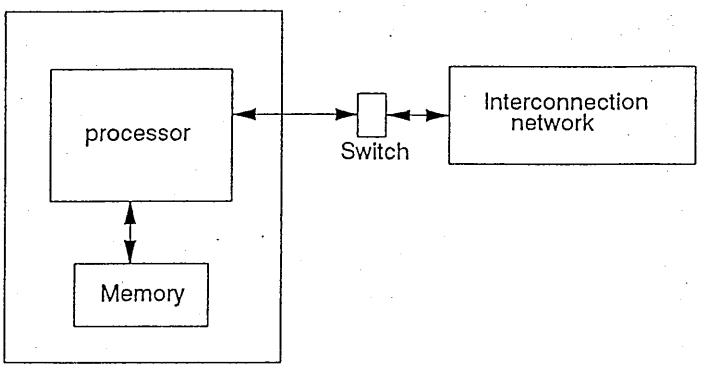



Figure 2.3: A unit in a multicomputer communication network

A range of communication architectures may possibly be used with these systems. This range extends from a shared bus in which all the processors of the system are connected to a single bus, up to a fully connected network, where every processor is connected to every other processor. The Ethernet is a well-known example of a shared bus and it will be described in this chapter.

Fault tolerance, which includes fault diagnosis, is of critical importance in distributed systems especially if they are highly integrated. In these systems, the large number of interconnected elements increases the probability of faults that can occur. Communication networks of these systems are normally high by structured (e.g. binary tree, mesh, hypercube, etc.), and hence have fault-tolerance capabilities due to the existence of redundant paths from any source of a message to any destination. In fact, for a communication network to be capable of tolerating single faults, two paths between any pair of source and destination that have no links in common must exist.

With the ever-growing complexity of computer systems a unique and highly desirable relation between fault tolerance and graph theory has been formed. The intention was to design cost-effective fault-tolerant computer networks. Thus, a number of fault tolerant communication architectures based on graph structures have been developed. We will refer to these architectures here as *graph networks*. Many graph networks are highly struc-

tured and possess attractive features such as low diameter, symmetry, simple routing, fault tolerance, etc. [Bhuy84,Chen90,Ghaf89, Prad82,Prad85,Prad86]. A set of graph networks has been developed and used in this work. Its definition and construction procedure as well as the general characteristics of graph networks will be presented in this chapter.

*Routing*, is the technique that is used to determine the path along which messages have to travel between a source and a destination. For a communication network to provide an efficient operation, an adequate routing algorithm is essential. The degree of difficulty of such an algorithm is strongly affected by the network architecture. The implementation of fault diagnosis as with any other activity in distributed systems involves a number of message transactions. The routing of these messages requires a choice of a particular path, which is normally characterized by the delay on the path and by the number of hops between the source and the destination. An efficient routing technique may have a great impact on successful diagnosis. In this chapter some of the routing mechanisms that are used in computer networks will be presented.

## 2.2 A Shared Bus Communication Network

The single shared bus is an important communication network, and it is widely used for connecting distributed systems due to its simplicity and cost effectiveness. The basic organization of a shared bus was shown in Figure 2.2.a. Different types of modules can be connected to the shared bus and if a number of these modules is designed to both transmit and receive data, only one is allowed to transmit on the bus at any one time, however, more than one module can receive. The physical structure of a single shared bus does not offer fault tolerance, therefore some design alternatives have been implemented to eliminate the potential bus failure problems. Examples of these designs are: the MIL-STD 1553 [Digi78] and IEEE standard 802.4 [IEEE85] buses. The Ethernet is another popular example and being accepted as an industry standard for local area networks by a number of independent vendors.

**Example 2.1:** Ethernet [Slom84]

Ethernet is a serial bus and it forms the basis for the IEEE 802.3 local area network standard. It uses a coaxial cable with data rate of 10 Mbits/s. The cable is divided into segments of length not exceeding 500m. The Ethernet can be used to connect a

maximum of 1024 modules over a maximum distance of 2.5 km. A cyclic redundancy check is used for error detection and wrong messages are ignored. There is only one path between any source and destination, hence the bus is not fault tolerant.

Among many methods of access control used in local area networks, the Ethernet uses the Carrier Sense Multiple Access with Collision Detection (CSMA/CD). This technique is also referred to as "listen before talk" and it is completely distributed throughout all modules, which have equal priority. Any module that wants to transmit must make sure that the bus is not in use by another module before starting its own transmission. The bus can not be used by more than one module at a time. It is possible, however, that two modules decide to transmit simultaneously, if they both find the network clear of traffic, such a case is referred to as *collision*. A transmitting module connected to the Ethernet is capable of detecting the occurrence of collision and therefore stopping its packet. The module, after some random interval, will retry the transmission again, and if it does not succeed, the retry continues up to 16 times, after which the module reports an error condition.

The packet format of the Ethernet is shown in Figure 2.4. Definitions and functions of the fields in this Figure are as follows:

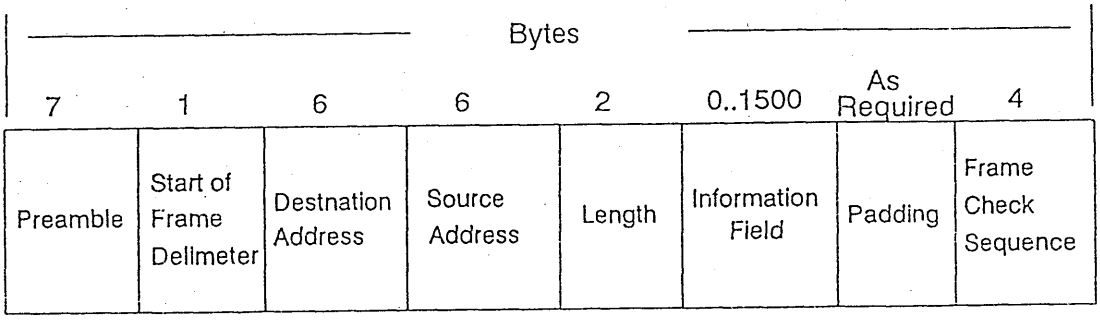



Figure 2.4: Packet format of the Ethernet

Preamble Field : This is a 4 bytes field. Its function is to synchronize each receiver module with the transmitter.

**Start of Frame Delimiter** : This is a one byte field with a sequence of alternating ones and zeros apart from the last two, which are ones. It indicates the start of the packet.

**Destination and Source Addresses** : Each of the two is a 6 bytes field containing the packet destination and source addresses. The destination address can be either for one module or as a multi-destination address. A single message can be sent to a group of modules by setting the first bit of the destination address to "1". If a message is intended for all modules, then a special address with all ones is used.

**Length** : This is a two bytes field. The value of its contents refers to the number of bytes in the information field of the packet.

**Information and Padding Fields** : The information field is specified by the standards to have a maximum of 1500 bytes and a minimum of 46 bytes. If the length of the information in the packet is less than 46 bytes then a number of extra bytes is added to the end of the information field in the padding field.

**Frame Check Sequence Field** : This field contains 4 bytes of a cyclic redundancy check code designed for error detection. The code covers the part of the frame starting at the destination address. If an error is detected the frame will be ignored.

The Ethernet is a very efficient bus, however, it should not be loaded above 50% as delays increase. Delays in this bus are of probabilistic nature, and unlike some other buses, it is not possible to define a value for the delay on a particular message in the Ethernet.

## 2.3 Graph Networks

### 2.3.1 Introduction

In designing a fault tolerant communication network, two main categories of fault tolerant techniques may be implemented. The first is not directly related to the architecture of the system, and its application does not involve modification in the network structure. The use of error correcting codes is an example of this category. In the second category, which we are more concerned about, fault tolerant techniques involve modifications in the architecture of the system. The implementation of multiple buses, adding extra communication links, and adding extra switches are some examples of this category.

Network architectures can be expressed in terms of the network diameter and the degree of its processors. *Network diameter* is the maximum number of communication links (hops) between a source of a message and its destination along the shortest path, while the *degree* of a processor is defined by the number of communication links connected to it. The design of a communication network is motivated by the minimization of both the network diameter (to reduce interprocessor distances), and the degree of processors (to provide a practical and cost effective network). These two issues are interrelated and they are considered whenever a graph network is designed. A graph network is normally of high connectivity, (the *connectivity* of a graph G is the minimum number of vertices, whose removal will disconnect G [Gibb85]). It is also of high structure and capable of supporting routing and broadcasting strategies, which are usually extendible to failure conditions.

In this section a class of graph networks will be presented, whose design has been developed and implemented in our work. The design is based on graph structures which we refer to as *H-graphs.*

### 2.3.2 The H-graphs

We consider the design of a class of graph networks that are both regular and homogeneous. These graphs are a generalization of the $D_{\delta t}$ graphs proposed in [Prep67]. A *regular* network is defined as the one with all its processors being of the same degree, while the *homogeneous* network is the one with all its processors being topologically identical. The underlying connection of the networks considered here is a single loop network $H_0$, which forms a cycle of $n$ processors each of degree 2. The single loop system is 1-fault diagnosable, according to the PMC model, and its diagnosis problem has been studied in [Prep67]. To simplify the design, we will assume that all processors are of the same type. We further assume that $H_0$ is represented by a directed graph in which all processors are labelled. One way for labelling is to assume that the processors are numbered 0 through (n-1) in the direction of the graph. A directed link in $H_0$, which connects processor $i$ to processor $i+1$ refers to a testing assignment in which processor $i$ has been assigned to test processor $i+1$. (All arithmetic is modulo $n$).

Although the single loop network allows for a very simple routing, the network has a large diameter and it is not fault tolerant. In H-graphs, the diameter of the network can

be reduced by adding an equal number of extra links to every processor in $H_0$, which will result in increasing the network connectivity and hence its fault tolerance. The H-graphs are defined by the following Formula :

$$H(n, r; k_1, k_2, ...k_r) \qquad \text{..... 2.1}$$

In this Formula :

$n$ - represents an arbitrary number of processors ($n \geq 3$).

$r$ - number of input(output) links in each processor. Output links represent test assignments of the processor.

$k_1$ ... $k_r$ - integers represent the number of hops in $H_0$ between any processor in the network and the processors which are connected to it through its output links. For consistency, these integers always appear in ascending order.

In order to form the underlying network $H_0$, the value of $k_1$ must be 1.

Example 2.2: For $n = 7, r = 2, k_1 = 1$, and $k_2 = 2$ we have the graph network H(7,2;1,2), which is shown in Figure 2.5.

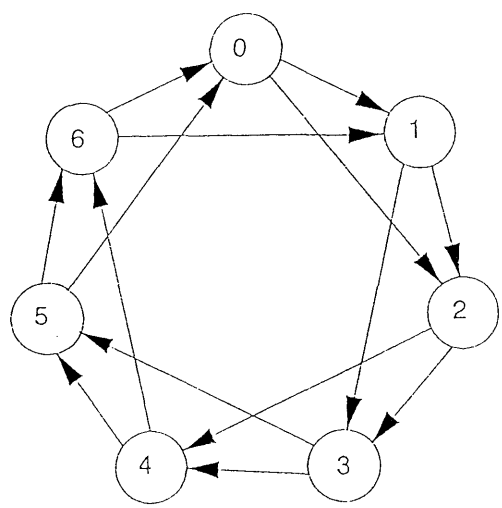



Figure 2.5: The H(7,2;1,2) graph network

The H-graphs are regular, homogenous and posses a cyclic symmetry. There is a number

of distinct graph networks that can be formed for a given $n$ and $r$, using Formula 2.1. For any graph network there may exist a number of equivalent networks, which are defined by the following Formula :

$$H(n, r; k_1 * m, k_2 * m, ....k_r * m) \qquad ..... \ 2.2$$

In Formula 2.2, $m$ and $n$ are relatively prime and all multiplications are mod $n$.

Example 2.3 : For $n = 5, r = 2, k_1 = 1$, and $k_2 = 2$ a graph network H(5, 2; 1, 2), which is shown in Figure 2.6, is formed by using Formula 2.1.

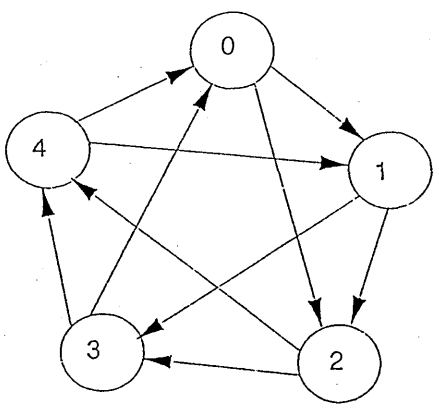



Figure 2.6: The H(5, 2; 1, 2) graph network

If we choose $m = 2$, a graph network H(5, 2; 2, 4) will be formed, which is shown in Figure 2.7.

The graph network in Figure 2.7 is equivalent to that of Figure 2.6. This equivalence can be made clearer if we consider that $H_0$ in Figure 2.7 to consist of the directed cycle *0-2-4-1-3* instead of *0-4-3-2-1* as shown in Figure 2.8.

In a similar way, the graph networks H(5,2;1,3) and H(5,2;3,4) can be shown to be equivalent to H(5,2;1,2).

The only two possible permutations that are left from $n = 5$ and $r = 2$ are H(5,2;1,4) and H(5,2;2,3), which are equivalent to each other. Moreover, in these networks each two

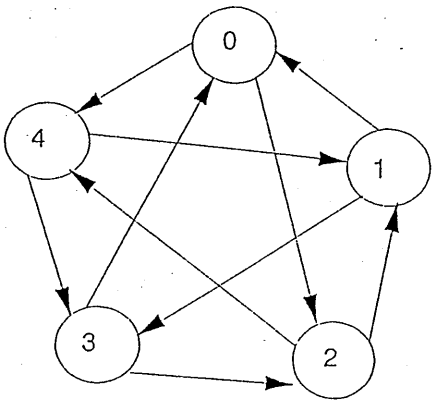



Figure 2.7: The H(5, 2; 2, 4) graph network

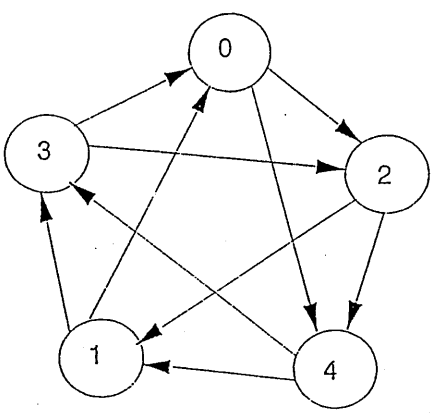



Figure 2.8: An equivalent H(5,2;1,2) graph network

adjacent processors are testing each other and for this reason such an architecture will not be considered in our work, where they are vulnerable to single processor failures. It is therefore possible to conclude that only the graph network H(5,2;1,2) is distinct.

Graph networks that are generated from the H-graphs can be characterized by the following features:

1. Both the regularity and homogeneity of processors in these networks are practically useful, where they allow for the design of a distributed system that is composed of inexpensive replicated processors.

2. As with some other proposed graph networks [Prad86], the resulting architecture can be considered as logical rather than physical and hence it may be used to represent task assignments. In fault diagnosis, task assignments are the scheduled tests of processors. Using graph networks, both the processors and their tasks can be represented as a graphical model, which does not necessarily fit the physical architecture of the system.

3. The graph networks proposed here are easy to construct and they have a general form, which can be extended to construct some other proposed graphs in an easier way as shown in the following example:

**Example 2.4** : The C-wrapped Hexagonal architecture is a class of graph networks that has been studied in [Chen90] and implemented in an experimental distributed real-time system [Dolt91]. The underlying connection of these networks is a hexagonal mesh which is similar to the one that was shown in Figure 2.2.j. The hexagonal mesh which was shown in that Figure is of degree 2.

In order to demonstrate the design procedure proposed in [Chen90], let $d$ be the dimension of a hexagonal mesh and let $n$ be the number of processors, which is given by: $n = 3d(d-1)$. Thus, for $d = 3$, $n$ will be 19. Each processor $i$ ($i = 0...n-1$) has six neighbours, which are given by:

$i+1$

$i+3d-1$

$i+3d-2$

$i + 3d(d-1)$

$i + 3d^2 - 6d + 2$ and

$i + 3d^2 - 6d + 3$

Where all calculations are mod $n$.

For every processor $i$, these values need to be found in order to implement a correct labeling and hence wrapping. The way in which a hexagonal mesh of dimension 3 is being labeled and wrapped is shown in Figure 2.9.

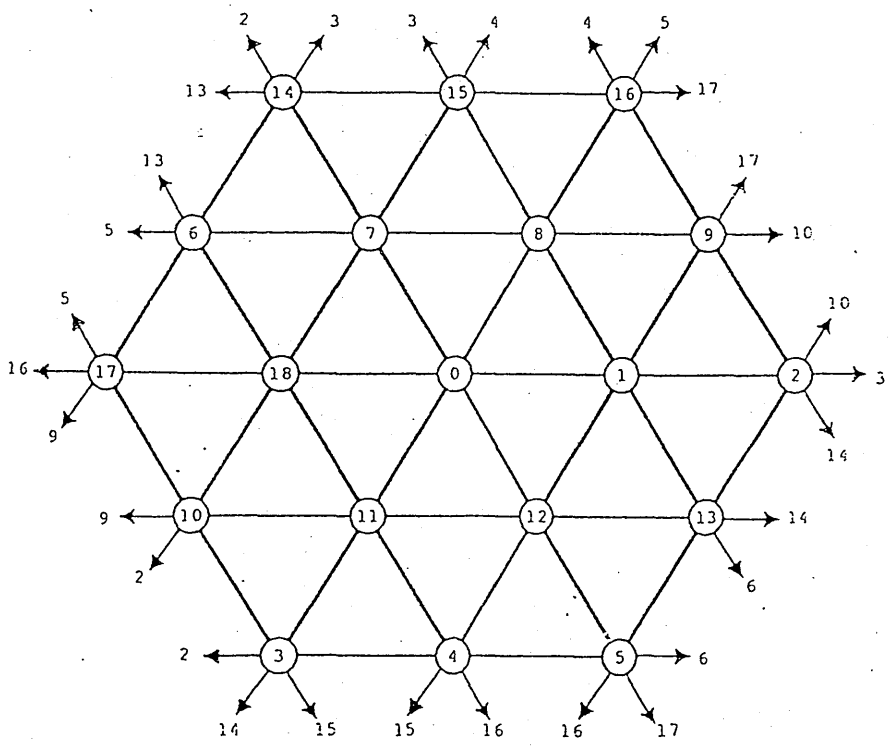



Figure 2.9: A hexagonal mesh of dimension 3

By using the H-graphs, and considering the resulting graph to be undirected, the wrapped hexagonal mesh of dimension $d$ can be constructed using Formula 2.1. The parameters of this Formula will have the following values: $n = 3d(d-1), r = 3, k_1 = 1, k_2 = 3d-2$ and $k_3 = 3d-1$. Thus, for $d = 3$ a graph network H(19,3;1,7,8) is formed, which is shown in Figure 2.10. The network shown in this Figure is similar to the C-wrapped

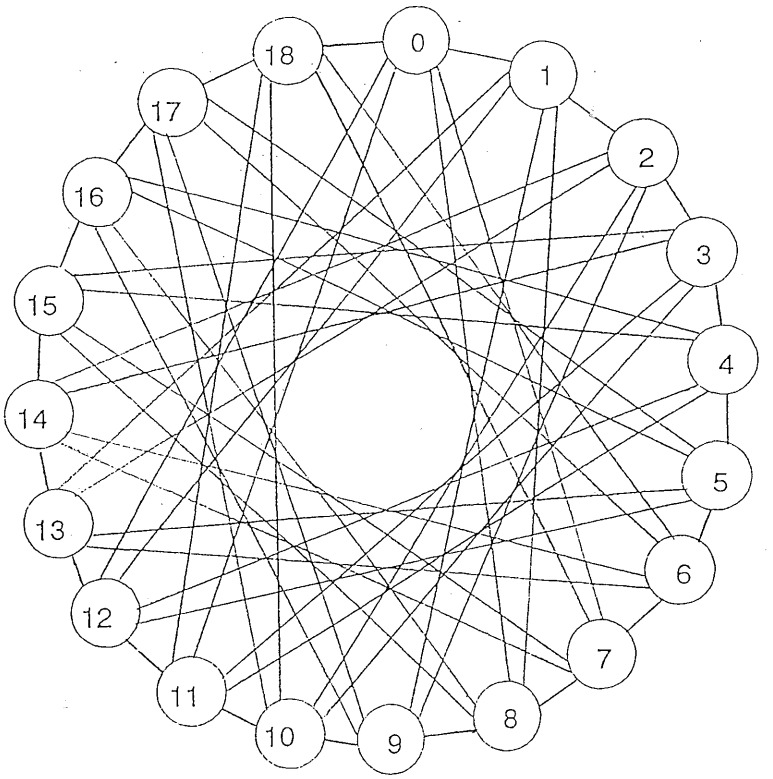



Figure 2.10: H(19,3;1,7,8) which is similar to a C-wrapped hexagonal mesh of dimension 3

hexagonal mesh shown in Figure 2.9. However, its construction procedure is shown to be much easier.

## 2.4 Routing

### 2.4.1 Introduction

We have mentioned in the introduction to this chapter that, in order to perform a successful operation, a communication network requires an adequate routing algorithm. The function of this algorithm is to determine the path along which messages have to travel between a source and a destination, and to maintain and update the information on which the choice of the path has been made. In a single shared bus or a loop architecture there is only a single path for all communication, hence a very simple routing is required. In a communication network, which offers more than one path, however, the source need

to decide upon which path should be used.

The implementation of a certain routing strategy is affected by the network architecture. In a star connected network, for instance, all traffic must pass through a single central processor. In a system connected by a single loop network any destination can be reached by any source either way via the loop. Meanwhile, for a more complicated architecture, like the rectangular mesh, each processor in the network needs to know how to direct its traffic. These information can be provided in the form of a routing table at each processor. A relatively simple routing algorithm is required for a tree connected architecture, where it might be enough to indicate whether the traffic is traversing the tree up or down. A completely connected network is the one with every processor being connected by a dedicated point to point link to every other processor. These networks, which are not favorable because they are very expensive, can give very high reliability if a suitable routing algorithm is implemented to make use of the alternative routes available.

In real life applications, many communication networks do not have regular architectures, and if they are originally regular, they might lose their regularity after the failure of a processor or a communication link. For these applications, routing algorithms need to be adaptive and the routes should be dynamic, to enable the routing algorithm to avoid the formation of message bottlenecks.

Many routing techniques are used in both circuit and packet switched networks. In a circuit switched network the routing algorithm selects the route, which will then be used for the whole duration of transmission. In a packet-switched network, however, the algorithm may either determine individually the routing of each data packet or else set up a route to be followed by a sequence of packets. In the following section, some of the routing techniques that are used in packet switched networks are defined.

### 2.4.2 Possible Routing Techniques

**Random Routing** : In this technique, messages does not progress systematically towards their destination. Instead, they travel randomly. The technique is simple and the routing process does not depend on the knowledge of the network.

**Flooding** : This is a simple routing technique, and its implementation does not depend on previous knowledge of the network architecture. Each processor that has a message to be transmitted is required to forward this message to all its output

links. This is then repeated for every processor that receives the message. In order that the message does not pass through the system continuously it must be damped. One way to achieve this is to include a count in each stage, which is incremented each time the message passes from one processor to the other. The message should then be destroyed if the count exceeds a certain value.

In this routing technique, the communication network is not being utilized efficiently and this is a disadvantage. However, it is advantageous throughout its ability to survive processor and link failures, and in that at least one copy of the routed message will reach its destination by the shortest path and therefore in the shortest time.

**Directory Routing** : In this technique a routing table with a series of possible routes is maintained in each processor. The technique is quite efficient for less loaded networks but when the network start to be loaded it will be very inefficient.

If all the possible routes to every destination are to be stored, a very big routing table or even more than one table is required, and this is not always suitable. As an alternative, every processor is provided with a single table which contains the identity of the neighbouring processors to which a message should be passed to reach a given destination.

**Adaptive Routing Techniques** : These techniques are designed to allow the use of multiple routes, taking into account failures in the network and accepting the addition or deletion of processors. Their design problem lies in the updating of routing information due to the distributed nature of the network. One of these designs is called *Isolated Adaptive Routing.* In this technique, processors make routing decisions according to the information that is available locally at each one of them. The required information is provided as a routing table, and the routing algorithm is programmed to make a choice between alternative routes. When a message arrives at a processor it is retransmitted as soon as possible using the route of least weight. The weight of the route is calculated from the length of the transmission queue and the number of communication links(hops) required to reach a destination by using this route.

In a rather more popular adaptive routing technique called *Distributed Adaptive*

*Routing*, the routing algorithm aims to find the route with the least delay, where each processor is provided with a table of routes giving least delay to each destination. As in the previous technique, the delay on the route is assessed by the number of hops and the queue length. Routing tables are continuously updated, where, in a synchronous manner, processors exchange with their adjacent neighbours their updated information. Routing tables at the relevant processors are recalculated and updated with new values for delay time. Considerable amount of messages are required to perform the exchange process, which normally affects the network performance. As a partial solution to this problem, it was proposed [Davi79] to make the exchange asynchronous rather than synchronous, where only the processors with significant change are to be involved.

**Hierarchal Routing** : In a large computer network, the implementation of some of the previous routing techniques may be difficult, where the increase in the size of the network may involve using different standards or may be different maximum packet sizes. Moreover, in these networks, if a routing technique which requires the use of routing tables is used, then a very large table must be provided to each processor. The hierarchal routing has been developed to adapt with the nature of these networks. The implementation of this technique involves a restructuring of the network in a hierarchal way, where groups of adjacent processors are assigned to a number of clusters and the routing table has only to contain addresses of these clusters. The network may be further divided so that a number of clusters can be designed to belong to a higher level cluster.

A message that is routed by this technique need to follow the hierarchy, which means that a longer path may be required than in a non-hierarchical system , where paths are separately specified.

# Chapter 3

# Fault Diagnosis in Fully Distributed Systems

## 3.1 Introduction

There has been considerable research on constructing computing systems that are composed of very large networks of distributed processing nodes [Bhuy84, Sull77, Swan77, Witt78]. Processing nodes in these systems are interconnected via communication paths (e.g., busses, fibre optic links, etc.). There is often no central facility in these systems to provide control or coordination among the processors. In fact, for very large systems, such a facility is often not feasible, which necessitates the employment of distributed control among the processing nodes themselves. In this and next chapters the term *distributed systems* will be used to refer to this type of system.

Due to the large number of processing nodes employed, failures in these nodes must be expected to occur, which if allowed in the system without checking, the whole system will be affected. Thus, a facility for handling these failures is required. Its job will be to detect failures and somehow remove or isolate faulty nodes from the system. This facility will be considered to be distributed throughout the system, where it will form part of each processing node.

By this consideration, the node, which needs to know the condition of other nodes in the system, will be obliged to depend only on analysis of information obtained through normal communication links. Neighbouring nodes, which share direct communication links are probably capable of performing tests among themselves and therefore obtaining information directly via these links. However, the node must rely on indirect knowledge about a non-neighbouring node, which has to communicate indirectly.

It is essential for these systems to have the property that each fault-free node must be independently capable of achieving correct diagnosis of all failures in the system.

Facilities in the form of distributed diagnosis algorithms have been proposed to assist the node in meeting this requirement.

## 3.2 Diagnosis Model

In earlier models, fault diagnosis and the activities that follow such as repair and reconfiguration, are performed either under control of a central facility or by means of some outside intervention. In our work we will consider a diagnosis model, which is based on a model proposed by Kuhl [Kuhl80c]. In this model, fault diagnosis is performed by the nodes of the system themselves. The model also involves the flow of diagnostic information through the system network.

According to this model, a distributed computing system is defined by two graphs called the System Communication Graph or simply the *Communication Graph C*, and the System Testing Graph or simply the *Testing Graph* $T_s$.

*Definition 3.1 : The Communication Graph C.* This is a unidirectional graph defined as $C = \{V(C), E(C)\}$ with $V(C)$ being set of vertices representing system nodes and $E(C)$ a set of edges representing the communication links. An undirected edge between two nodes $P_i$ and $P_j$, denoted as $(P_i - P_j)$, in the communication graph means that a direct communication link exists between two processors in the system, which are referred to as *neighbours*.

*Definition 3.2 : The Testing Graph* $T_s$. This is a directed graph defined as $T_s = \{V(T_s), E(T_s)\}$ and a subgraph of the communication graph C. It is often considered to have the same set of vertices as the communication graph (i.e., $V(C) = V(T_s)$), but a different set of edges $E(T_s)$. A directed edge $(P_i \rightarrow P_j)$ in this graph, means that node $P_i$ is assigned to test node $P_j$.

Symmetric invalidation of tests is assumed in this model, and faults occurring in the system are assumed to be permanent.

The communication graph of a distributed system will be identical to its testing graph, if all the communication links are used for testing. In such case one graph may possibly be enough to define the system. However, this is only a special case, and for a general model the communication and testing graphs need to be considered as two separate entities. The choice of a testing graph for a certain system may be influenced by one or more of

the following:

1. There might exist a number of edges in the communication graph, whose removal will not reduce the connectivity of the graph especially if these graphs are irregular. Assigning tests to these edges will not increase the diagnosability of the system, therefore they are often eliminated in the testing graph.

2. The testing process is time consuming in terms of the testing load placed on the processors and the number of diagnosis messages produced to perform the diagnosis. To reduce these two, the number of tests has to be minimized by constructing a small testing graph, which can offer a certain degree of diagnosability. The degree of diagnosability of a graph is related to its connectivity, and for certain testing graphs this could be lower than that of the communication graph. For a given system there is a tradeoff between keeping the testing graph small and obtaining higher degree of diagnosability.

3. Diagnostic information about the processors of a system are considered to flow through its testing graph. The time required for these information to reach a node ($P_i$), that has to diagnose the condition of another node ($P_j$), is related to the shortest path by which the information are received. Testing graphs are to be chosen so that the paths on which diagnosis messages travels, are as short as possible.

Considering points 1 and 2, a graph network $H(n, r; k_1 ... k_r)$, constructed using Formula 2.1, which was introduced in section 2.3, for example, can have a number of different testing graphs for a given $n$. A small testing graph with reduced number of tests can be produced by reducing the value of $r$, while by increasing the value of $r$, a testing graph with higher degree of diagnosability can be produced. The length of the paths of diagnostic information can be minimized by adjusting the values of $k_1$ ... $k_r$.

## 3.3 Self Diagnosability

A measure for t-fault self diagnosability was proposed by Kuhl [Kuhl80c] to meet the requirement of system level diagnosis in distributed systems. The proposed diagnosability measure was based on the following assumptions :

*Assumptions 3.1*:

- Tests can only be conducted among neighbouring nodes.

- Faults are permanent.

- Test results reflect the actual condition of the system, (i.e., tests will always fail if the node being tested is faulty and always pass if there is no fault).

- Exchange of diagnostic information is carried out by messages passed between neighbours.

- Fault-free nodes always handle messages correctly,while there is no certainty about messages handled by faulty nodes.

Based on these assumptions, the following definition has been set,

*Definition 3.3*: A distributed system with communication graph $C$, and testing graph $T_s$ is said to be *t-fault self diagnosable* for a set of $t$ or fewer faulty nodes, if and only if, each node in the system is capable of reliably diagnosing the condition of all other nodes in the system, by means of test results being conducted through $T_s$, and by analyzing information contained in diagnostic messages received from neighbours.

Nodes that will be involved in a diagnosis procedure, which considers this definition, will face two important considerations; firstly nodes cannot rely on any diagnostic information they receive, since the condition of their sender or the nodes they pass through is not known. Secondly, in a typical distributed system, a node might receive a diagnostic information about the condition of another node via many paths, whose number might be very large and the information might be received correctly over some of them and corrupted by others.

An approach to simplify these two considerations and produce diagnosis was defined in the form of diagnosis algorithms.

## 3.4 Self Diagnosis Algorithms

A series of distributed algorithms were presented, which basically depends on Definition 3.3 of the measure of t-fault self diagnosability. In this section the sequence of their evolution will be considered, while introducing the main assumptions on which they are based. Although these algorithms differ in detail they are all based on the ability of a node to perform tests on some of its neighbouring nodes and sending test results back.

(i) algorithm SELF [Kuhl80c] :

This algorithm was proposed under the following assumptions :

*Assumptions 3.2*:

- Tests are to be conducted periodically in cycles, and in accordance with their assignment in the testing graph.

- No faults are assumed to occur during the testing cycle.

- Since tests are to be conducted in a single instant in time, their outcome will represent a static condition of the system.

- A node may rely on information received from a neighbour only after conducting a test on it and being certain that it is fault-free.

- Test results are stored in two sets $D_0$, to contain tests which pass, and $D_1$, to contain failed test results. These two sets are assumed to represent the diagnostic messages, whose purpose is to exchange diagnostic information between nodes.

(ii) algorithm SELF2 [Kuhl80c] :

Algorithm SELF2 is an extended version of algorithm SELF with some additional assumptions.

*Assumptions 3.3*:

- It is assumed that individual nodes might perform their test duties according to some sort of local schedule.

- Due to the dynamic behaviour of the system the diagnostic procedure should be viewed as a constantly evolving event rather than a set of test results calculated at discrete time intervals.

- The latest diagnostic information, received at a node must be treated with suspicion. It should be buffered locally until being verified by another node.

- There is only one type of diagnostic message. The message is used to exchange diagnostic information between nodes and is assumed to contain an integer $i$ in the range 0 through $n$. The value of this integer indicates that the source of the message has found node $i$

faulty.

(iii) algorithm SELF3 [Kuhl81] :

The ability of the system to directly communicate in a reliable way might be affected by a failure in a communication path. Hence, it is reasonable to require that nodes of a self diagnosable system are capable of diagnosing such cases. Therefore the definition of self diagnosability was extended to allow for failures in both the nodes and communication links of a distributed network.

A new definition for self-diagnosability was given when introducing this algorithm. It is an extension of that given in algorithm SELF2. In addition to Assumptions 3.1,3.2 and 3.3, the following are assumed :

*Assumptions 3.4*:

- When a node detects a fault, it will not be able to decide by itself whether there is a real failure in the node or the detection is due to a faulty link. Hence, an additional diagnosis message is added in order to inform the system that the failure was due to a communication link. Routing of these messages is the same as those used to indicate a node failure.

- If a node faces difficulty in communicating with a neighbour, which it cannot test, then this node is allowed to indicate the failure of its neighbour to the system, so that the testers of the faulty node can perform a test. The test result will then be used to decide whether the tested node or the communication link is really faulty. In fact, this practice will help in diagnosing failures affecting communication links other than these included in the testing graph.

- As in algorithm SELF2,this algorithm views the diagnosis as a dynamic process, which continually evolves in time. However, it is not sensitive about time and it contains no limitations about the arrival times of messages.

- There are two main diagnostic messages, one is to inform the system about a node failure and the other is to indicate a link failure.

- $[P_r$ by $P_q]$. This message is issued by node $P_q$ if it tests node $P_r$ and finds it faulty, or if node $P_q$ faces difficulty in communicating with node $P_r$.

- $[(P_q \rightarrow P_r)$ link]. This message means that a fault in the link between $P_q$ and $P_r$ has been diagnosed.

(iv) algorithm NEW-SELF [Hoss84] :

This algorithm is based on the fundamental assumptions of algorithm SELF and the extensions in algorithms SELF2 and SELF3. A few further assumptions are considered by this algorithm.

*Assumptions 3.5*:

- The algorithm considers the diagnosis as a dynamic process in a more obvious manner, where time stamps in the form of sequence numbers, which increase successively are assumed to be appended to messages. Furthermore, the algorithm is designed to allow repaired or replaced facilities back, and for admitting new nodes and/or links into the network.

- In order to increase a reliable flow of diagnostic messages a node should not trust newly received messages. It is assumed that a node may accept a diagnostic message from its neighbour and temporarily stores it in a local buffer, provided that, this neighbour has been found fault-free, when it was last tested by the node. After testing its sender again, and making sure that it is fault-free, the message should be fully accepted.

- The diagnosis of faulty links is considered in this algorithm.

- Diagnostic information are exchanged between nodes using five types of diagnostic messages.

- $[nil, P_q(t_q), P_r]$. Node $P_q$ issues this message at time $t_q$, either because it tests node $P_r$ and finds it faulty, or because it has difficulty in communicating with node $P_r$.

- $[P_q(t_q), P_q(t_q), P_r]$. This message means that node $P_q$ has tested node $P_r$ and finds it fault-free.

- $[P_q(t_q), P_k(t_k), P_r]$. This message also means that node $P_q$ has tested node $P_r$ and found it fault-free. In this case, however, node $P_r$ has recovered from a fault that was previously reported by node $P_k$.

- $[link, P_q(t_q), P_r]$. This message is issued when node $P_q$, which is a neighbour of node $P_r$ but not a tester, faces difficulty in the communication with node $P_r$.
- $[link-up, P_q(t_q), P_r]$. If node $P_q$, which has issued a message similar to the previous one, is no longer facing difficulty in communicating with node $P_r$, it will issue this message.

## 3.5 Self Diagnosability Based on Self Testing System Nodes

Novak et al [Nova87] have presented the problem of incorporating the testing process in systems composed of self testing nodes. There was no specific algorithm proposed for the diagnosis but it was shown, that it can be viewed in the scope of the existing distributed diagnosis algorithms.

One of the proposed ways of practically carrying out tests is to apply a controlled stimuli and observe the response [Prep67]. In this way, a considerable amount of stimuli and responses will be required for achieving a reliable test, and this has to be in the form of transmitted information. If the nodes involved are self testing and only the result of such test is transmitted, then an important saving in the transmission efficiency of links will be obtained.

Similar to the way assumed in the previous algorithms, the diagnostic information will traverse the testing graph and nodes interchange these information until a status of a complete diagnosis among the system nodes is reached. For a distributed system to reach this status, it needs to go through a series of events. This fact emphasizes the requirement, that a distributed system must provide a clear ordering of events.

## 3.6 Some Implementations of Self Diagnosis Algorithms

In line with the rapid increase in the use of local area networks, the need for implementing fault diagnosis in their processors has emerged. The theory of system level fault diagnosis, that is based on a considerably large body of theoretical results, is an important step in this direction, but it has yet to be applied and exploited in real systems.

Two distinctive research works appeared in literature, which were facing towards a practical implementation of theoretical system diagnosability results to a real distributed network environment [Grif86] and [Bian90] . Both of them applies algorithm NEW-SELF [Hoss84]. This algorithm has some interesting features, where it allows for different net-

work topologies, and it is also able to distinguish between faults in processing nodes and communication links.

Griffith [Grif86], used this algorithm to propose a design for a fault tolerant distributed computer system for automotive applications. The notion of processing node given in the algorithm was defined in this work to consist of processors and application interfaces. A triangular geometry was proposed for network topology both at the node level and for the internal architecture of the processing node. Software was designed, however no results were given.

NEW-SELF was also considered by Bianchini et al [Bian90]. It was applied to a network of multiple multi-user workstations running the UNIX operating system and communicating via an Ethernet local area network. A number of assumptions and modifications were found necessary in order to adapt the theoretical results to actual systems. These modifications were contained in a modified version of algorithm NEW-SELF defined as EVENT-SELF. This algorithm was shown to produce fewer messages during routine scheduled testing when no new faults are reported, but higher message densities when a fault is detected than that of NEW-SELF. It was assumed that, while the system is not diagnosing a fault situation, then it should only be assigned to execute a minimum number of tests, which allows for specific level of diagnosability . However, when a fault is detected all other tests of the faulty node have to be initiated.

## 3.7 Modified Algorithm SELF3

The implementation of algorithm SELF3 in a distributed system may involve a temporary misdiagnosis of some fault-free nodes as faulty if failures in communication links occur. For this reason, a modified version of this algorithm has been presented by Hosseini et al [Hoss88]. However, we have noticed that algorithm NEW-SELF, which has been implemented in [Bian90] and [Grif86] has the same deficiency. Therefore, modification of algorithm SELF3 could be made to deal with the shortcome of algorithm NEW-SELF as well. In the following, the manner in which a temporary misdiagnosis might occur in both algorithms, will be shown.

Consider a distributed system part of whose testing graph is shown in Figure 3.1. If in this system a test of a node $P_r$ by a node $P_q$ has failed, while in fact $P_r$ is fault-free but the communication link connecting $P_r$ and $P_q$ is faulty. The node $P_q$ will issue a message

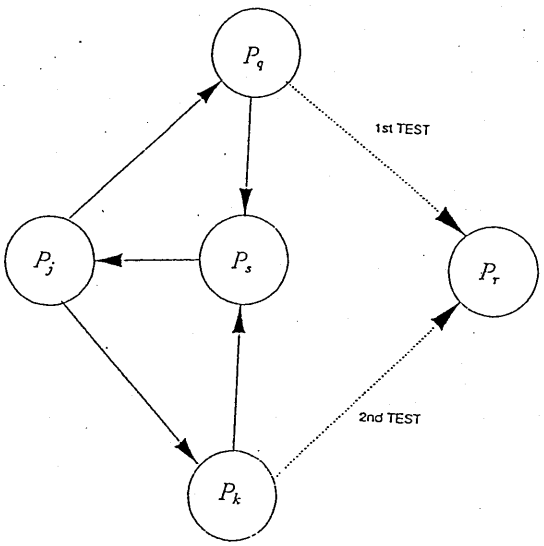



Figure 3.1: Testing graph of a distributed system

of the form [ $P_r$ by $P_q$ ] when algorithm SELF3 is applied or of the form $[nil, P_q(t_q), P_r]$ if algorithm NEW-SELF is applied. This message will traverse the testing graph, in an opposite direction to the arrows,from node $P_q$ to other nodes. A node, which receives such message will, for the time being,consider the node $P_r$ as faulty until another node $P_k$, which is a fault-free tester of node $P_r$ receives the message. This node will conduct a test on $P_r$, where it will be found fault-free. A new message of the form [ $(P_r \rightarrow P_q)$ link ] if algorithm SELF3 is used or of the form $[link, P_q(t_q), P_r]$ if algorithm NEW-SELF is used, will be issued. This diagnostic message will also be routed through the system using the testing graph. Nodes, when receiving this message will realize that the condition of node $P_r$ has been misdiagnosed, and hence they will correct this and update their diagnostic information accordingly.

*Modified algorithm SELF3* was proposed to deal with this confusing behaviour in the diagnosis process, and to avoid the occurrence of temporary misdiagnosis, that has been demonstrated. As with the aforementioned algorithms, this one is designed for *homogenous systems* , which are a class of distributed systems with each node being capable of performing tests on some other nodes in the system. This algorithm will be looked at in more detail since it has been selected to be used as a starting point for constructing

the simulator. Some of the terms and notation of [Hoss88] and [Kuhl81] will be used to introduce the diagnosis algorithm.

### 3.7.1 Preliminaries

Before introducing the diagnosis algorithm, the following need to be defined :

*Definition 3.4*: *Link Failure Domain (LFD)* . Any single source of failure of communication links in the distributed system forms a link failure domain.

*Definition 3.5*: *Link Failure Domain Set (LFDS).* In a distributed system, this represents the set of all possible link failure domains, and is denoted by $D_s$.

From these definitions, it is therefore possible to represent a distributed system as a triple $(C, T_s, D_s)$.

Let $D \subseteq D_s$ and $P_r \in V(C)$ in a system $(C, T_s, D_s)$ then,

*EDGES(D)* - Set of communication paths lost due to the link failure domain in D.

*INVAL-BY(D)* - Set of tests (edges of $T_s$) invalidated due to the failures in D.

*TESTERS-OF($P_r$)* - Set of nodes $[P_q \in V(T_s) \mid (P_q \rightarrow P_r) \in E(T_s)]$.

*TESTED-BY($P_r$)* - Set of nodes $[P_q \in V(T_s) \mid (p_r \rightarrow P_q) \in E(T_s)]$.

### 3.7.2 The Diagnosis Algorithm

The modified algorithm SELF3 assumes that every node $P_i$ in the system has two sets $ND-FLUR_i$ and $LNK-FLUR_i$. The elements of $ND-FLUR_i$ are faulty nodes in the system, while the elements of $LNK-FLUR_i$ are faulty communication links between $P_i$ and the nodes with which it has direct communication links. When a node $P_q$ is assigned to test another node $P_r$, they are called *tester* and *testee* respectively. The algorithm employs the following forms of messages.

[ $P_r$ by $P_q$ node ], this messages is referred to as a 'broadcasting message', and it means that node $P_q$ has determined that node $P_r$ is faulty.

[ $P_q - P_r$ link ], this is also a 'broadcasting message' and it means that the direct communication link between $P_q$ and $P_r$ is faulty.

$[?, T, P_q, P_r, P_q]$, this is called 'interrogation message'. Whenever a node $P_q$ tests a node $P_r$ and $P_r$ fails the test then $P_q$ will interrogate all its fault-free testees $P_s$'s (i.e., the nodes that have passed the test performed on them by $P_q$ ) about the condition of $P_r$ ,

by sending an interrogation message, of the form shown, to $P_s$. The *set* $T$ is used in this type of message to insure that they traverse the testing graph only through acyclic paths. Its initial contents are $T = [P_q \bigcup P_s \in TESTED - BY(P_q)]$. A node that is contained in $T$ should not be re-interrogated by another node, receiving a message comprising this set, about the condition of an accused node.

$[YES, P_q, P_r, P_s]$, this message is for 'transmitting a test result'. When node $P_s$ receives an interrogation message, it will conduct a test on $P_r$ if it is a tester for this node, and if $P_r$ passes the test then this message will be sent.

$[NO, P_q, P_r, P_s]$, this message is also for 'transmitting a test result'. It is to be sent back in two cases, first if node $P_s$, which has received an interrogation message regarding the condition of $P_r$ is a tester and $P_r$ fails the test by $P_s$. Second, if $P_s$ is not a tester of $P_r$ and it has no fault-free testees $P_t \in TESTED - BY(P_s)$ such that $P_t \notin T$.

Alternatively, if $P_s$ is not a tester of $P_r$ but it has some fault-free nodes $P_t \in TESTED - BY(P_s)$ and $P_t \notin T$, then $P_s$ will set $T = T \bigcup [P_t]$, and then interrogate each node $P_t$ regarding the condition of $P_r$ by sending a message $[?, T, P_q, P_r, P_s]$ to $P_t$.

Consider a node such as $P_s$, that has interrogated a number of nodes (say $P_t$). If node $P_s$ receives a message of the form $[YES, P_q, P_r, P_t]$ from at least one of the nodes $P_t$, then it will pass a similar message $[YES, P_q, P_r, P_s]$ to node $P_q$, which is its interrogator. However, if node $P_s$ does not receive at least a "YES" message from any of the nodes $P_t$, then it has to wait until it receives a message $[NO, P_q, P_r, P_t]$ from all of them. A similar message of the form $[NO, P_q, P_r, P_t]$ will then be sent to its interrogator $P_q$. These actions, that are described at node $P_s$ will be followed by every other node, that has been interrogated.

At node $P_q$ (the initial tester), a different set of actions are required to be taken against the receipt of a test result message. If at least one message of type "YES" has been received at node $P_q$ from any of the nodes, that it has interrogated about the condition of $P_r$, then it recognizes that $P_r$ is fault-free but the communication path between $P_q$ and $P_r$ is faulty. This information will be kept locally in $LNK - FLUR_q$. Otherwise, if $P_q$ receives messages of type "NO" from all the nodes that it has interrogated about the condition of $P_r$ then it will consider node $P_r$ faulty, hence a message of the form [ $P_r$ by $P_q$ node ] will be broadcasted to every one of its testers.

Whenever a node $P_v$ receives a message [ $P_r$ by $P_q$ node ] from a fault-free testee, it will consider $P_r$ to be faulty, and hence add $P_r$ to its list of faulty nodes $ND - FLUR_v$, and it will send a message to every one of its testers.

As will be described in the next chapter, a unified format for the aforementioned messages are used in the simulator and few more messages are added. The operation of the algorithm will be further clarified in chapter 5, when a number of examples are presented.

The diagnosability of a system, which employs this algorithm is related to the connectivity of its testing graph $T_s$, whereas the connectivity of the testing graph is related to the minimum number k($T_s$) of nodes and communication links, whose removal will disconnect $T_s$. The following Theorem will be stated without proof [Hoss88].

*Theorem 3.1*: A system $(C, T_s, D_s)$, which employs modified algorithm SELF3 is $t_{n,l}$-fault self-diagnosable, if for its testing graph $T_s$, k($T_s$) $\geq t_{n,l} + 1$.

In such a system, each node can correctly identify all faulty nodes and communication links between itself and its fault-free neighbours, provided that no more than $t_{n,l}$ nodes and LFDS have failed.

## 3.8 Modified Algorithm SELF3: A Comparison with The PMC Model

It was mentioned in chapter 1, that referring to the early days of research in the theory of system level fault diagnosis, the PMC model, which is the fundamental model in the area was considered to be not applicable to actual systems, and therefore, a number of modifications and models were proposed. This model, however, has laid the ground work for later research, and most of the work in the area has focused on its generalizations. Symmetric invalidation of tests, Figure 1.1.a, was assumed in this model and the main results obtained in the work were given in Theorems 1.1-1.4. In this section, the diagnosability of a system and the tests required for performing diagnosis are compared, when the PMC model (diagnosis with global observer) and modified algorithm SELF3 (distributed fault diagnosis) are considered.

**Example 3.1:** Consider the H(7,2;1,2) graph network shown in Figure 2.5. This system, which may represent any H(n,2;1,2) graph network, is one-step 2-fault diagnosable according to Theorem 1.1 of the PMC model and $1_{n,l}$-fault self diagnosable according to

| | Test Syndromes | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Faulty nodes | 01 | 02 | 12 | 13 | 23 | 24 | 34 | 35 | 45 | 46 | 56 | 50 | 61 | 60 |
| 0 | x | x | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | x | x | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 2 | 0 | 1 | 1 | 0 | x | x | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 1 | 1 | 0 | x | x | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | x | x | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | x | x | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | x | x |

Table 3.1: Single fault test syndromes for the H(7,2;1,2) graph network.

Theorem 3.1 of Modified algorithm SELF3. With respect to the PMC model, the syndromes of tests for single or double faults are distinguishable, while according to Theorem 3.1, the failure of more than one node or link will disconnect the system, therefore the system can only diagnose a single fault in a node or a link. As an attempt to reconcile modified algorithm SELF3 with the PMC model we will consider the case of a single node failure, since it is common. Table 3.1 contains the test syndromes for single faults in the H(7,2;1,2) graph network, which are based on the assumption of the PMC model. According to this Table, any single fault is diagnosable, since all test syndromes are distinguishable.

(i) Consider modified algorithm SELF3 and assume that node 1 accuses node 3 (i.e., 13 = 1), where node 3 is faulty. Node 1 will interrogate node 2, which is a fault-free testee of node 1 (12 = 0) and node 2 tests node 3 and finds it faulty (23 = 1). The test result will be sent back to node 1 and node 3 will be considered faulty. Figure 3.2 shows the tests conducted during the diagnosis of node 3 using modified algorithm SELF3.

If we compare these test results (i.e., 12 = 0, 13 = 1, 23 = 1) with the syndromes in Table 3.1, we can see that they are enough to distinguish a fault at node 3.

(ii) Consider modified algorithm SELF3 again and assume that node 1 accuses node 2,

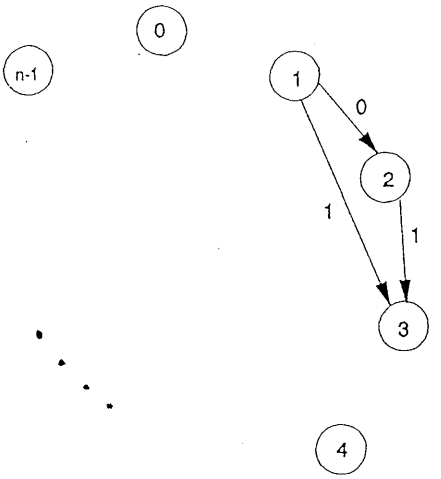



Figure 3.2: Tests conducted when node $i$ accuses node $i+2$ in the H(n,2;1,2) graph network for $i = 1$.

which is faulty(i.e., $12 = 1$). The interrogations will continue until node 0 is reached, which is a fault-free tester of node 2. Node 0 will conduct a test on node 2 ($02 = 1$) and then send the test result back. Node 2 will then be diagnosed as faulty. The test results required to evaluate the condition of node 2 are illustrated in Figure 3.3. From which the test results ($12 = 1$ and $02 = 1$) are in fact enough to distinguish the syndrome corresponding to a fault at node 2 in Table 3.1.

From this example we conjecture that; in general, if only node failures are considered, then the system which is $t$-fault diagnosable on the PMC model is only $(t-1)$-fault diagnosable using modified algorithm SELF3.

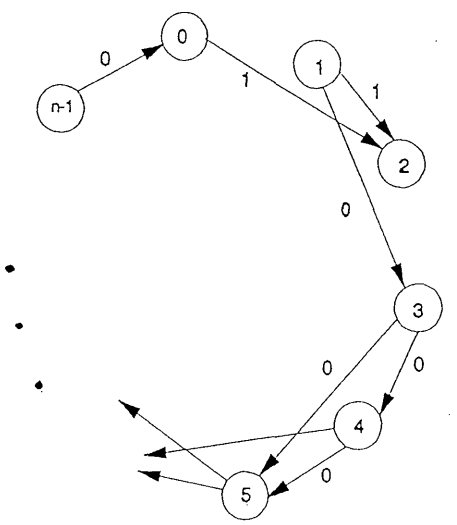



Figure 3.3: Tests conducted when node $i$ accuses node $i+1$ in the H(n,2;1,2) graph network for $i = 1$.

# Chapter 4

# Description, Structure and Input Data of The Simulator

## 4.1 Introduction

Simulation often provides the only performance evaluation tool for complex systems. Simulation of these systems is often distributed over a number of computers to reduce the programming effort and to improve the computation time. This type of simulation is referred to as *distributed simulation* [Righ89,Vrie90].

Simulation may have various objectives depending on what benefits are intended to be obtained from its implementation, and this issue has a significant impact on how a simulation should be designed. For instance, if a better understanding of behaviour is required, then a controllable simulation with adjustable initial conditions and well presented output should be among the main features of its design.

Although distributed simulation of a distributed system is possible - it would in fact be an imitative process - the choice has been made to simulate the distributed system using a single processor. This is because a range of algorithms have to be investigated and it is easier to collect the performance measurements together (e.g., total number of messages) when a single processor is employed. Also, parameters such as initial conditions, number of processors and connectivity are more easily changed when the simulation is performed on a single processor.

In this chapter a description of the simulator, that has been designed, implemented, and used in this work is presented.

## 4.2 System Modelling

### 4.2.1 The System Model

We consider a *physical system*, which is fully distributed and composed of a number of *processors*. These processors are communicating with each other exclusively by passing messages. We are concerned with simulation of this physical system, therefore we assume a *simulated system* and refer to it as *SIM*. Each processor in the physical system will be referred to in SIM as a *node*.

SIM will be simulated by a single processor and therefore it can only carry out one action at a time. It operates by simulating one of the processors of the physical system for the processing of one complete message. It then moves on to the simulation of one of the other processors for another message and so on.

### 4.2.2 The Simulation Algorithm

The simulated system SIM is asynchronous, and in order to achieve its asynchronism, it needs to calculate the simulated time $t_{ij}(k)$ at which message $k$ from processor $i$ to processor $j$ is received in the physical system. This time therefore has to be encoded as a part of the messages communicated through SIM. The operation of the physical system depends upon the order in which each processor receives messages. For SIM to simulate correctly, it must, for every node, process messages in the same order as the equivalent processor in the physical system.

The processing order in SIM, for any node $j$, requires the inspection of time stamps $t_{ij}(k)$. Therefore when SIM has to process the message $k$ from node $i$ to node $j$ then $t_{ij}(k)$ must be the earliest time stamp among messages queuing to be processed at node $j$. The selection of next processor to be simulated is carried out by SIM, by applying a three step simulation algorithm each time it executes a message and this is repeated until termination. These three steps are;

(i) Selection.

(ii) Processing.

(iii) I/O Operations.

For step (i) if we define;

$T_j$ - earliest time at which node $j$ is free to execute next message.

$t_{ij}(k)$ - time at which node $j$ receives a message $k$ from node $i$.

The next node to be executed is therefore given by :

$$[j | max\{T_j, min[t_{ij}(k)]\} \ is \ minimum]$$

or in another form,

$$[\ the \ earliest \ of \ \{\ processor \ free, \ earliest \ received \ message\ \}]$$

If there are more than one node for which this time is the same, the choice of which node to execute is arbitrary.

Steps (ii) and (iii) of the algorithm depends on the message type and the state of the system. Their implementation will be clarified in chapter (5).

### 4.2.3 System Messages

Since the simulator is designed to implement a fault diagnosis algorithm, only messages that are related to this objective are considered. However, in order to show that the program is not limited to diagnostic messages, a different type of message has been proposed.

The number of message types used in modified algorithm SELF3, which were presented in chapter (3), was found inadequate, hence their number has increased, and because they convey different information, a unified format has been chosen for the simulator with additional fields. One of these additional fields has been used for encoding simulating times as part of messages.

#### Message Format

Messages used in the simulator have the following format :

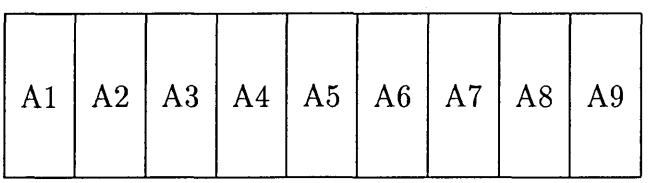



Figure 4.1: Message Format

Fields in Figure 4.1 are defined as follows:

- *A1 - message type*

- *A2 - initial tester*
- *A3 - accused node*
- *A4 - intermediate sender*
- *A5 - intermediate receiver*
- *A6 - execution flag*
- *A7 - completion flag*
- *A8 - time stamp*
- *A9 - set T .*

The function of most of these fields can be derived from the definition of A1, where the following types of messages are used;

**TEST - Test message.** In this message A4 is to conduct a test on A5.

**NDAC - Accusation message.** In this message A2 is accusing one of its testees A3 of being faulty.

**QUES - Interrogation message.** In this message A4 interrogates A5 about the condition of A3.

**PASS, FAIL - Test result.** In each of these messages, A4 has to route the test outcome of A3 to A5. The test might have been conducted by A4 or by some other node.

**NDFL - Broadcasting of node failure.** In this message A4 is to inform one of its testers A5 about the failure of A3.

**INFO - Information message.** This message has no role in the diagnosis, and it has been included in the set of messages to demonstrate the capability of the program for expansion, and hence accepting more types of messages. It can be used, however, to simulate processing tasks other than diagnosis in the system.

The rest of the fields, that have not appeared in previous definitions are :

*A6 - Execution Flag.* This is a two-valued parameter, which is (0) in messages waiting to be processed, and a value other than (0), defined as $COMP$ in a processed message.

*A7 - Completion Flag.* This is also two-valued, and is (0) in messages, waiting to be processed or waiting for a reply, and *COMP* in messages that have been processed and are no longer required.

*A8 - Time Stamp.* This is a positive number, whose value could either be equal to the local time of the node, that generates the message or to the simulated time at which the message reaches the receiving node.

*A9 - set T.* This field is exclusive to the interrogation message, and its size is related to the number of nodes in the system. The function of set T was described in chapter(3), and it will be further investigated in chapter(5).

### 4.2.4 Definitions and Assumptions

Definitions of some terms, that are used in this work are presented next.

*Difinition 4.1 : Diagnosis messages* . All messages that are generated in order to diagnose a fault in a system. This includes all types of messages defined in last section, apart from the message of type *INFO.*

*Definition 4.2 : Diagnosis time* . Time required for the system to reach a complete diagnosis status about a specific fault situation. This status is reached, when the relevant diagnosis information is received by all fault-free nodes, in case of node failure, or by the initial tester in case of link failure.

A number of assumptions about the simulator were made, these are listed below :

1. A node may or may not produce an output message(s) in response to an input message. A message could be produced and directed to the node itself.

2. Each node has a local queue, where messages relating to this node are enqueued. Moreover, each node has its own clock. The time in this clock represents the local or simulated time, and while simulating this is calculated as follows :

   (a) if the value of the clock is earlier than the time stamp of the message being processed then the clock is made equal to the time stamp of this message.

   (b) if the value of the clock is later than the time stamp of the message being processed, then the value of the clock is retained.

   After either (a) or (b), the value of the clock will be updated to its present value

plus the simulation time associated with processing of the last message, which corresponds to the delay introduced by the physical system performing equivalent processing.

3. The system is simulated with a shared-bus, and the use of such communication network involves a delay in transmission. Therefore the time stamp of a message at its sender is not the same time stamp of a message at its receiver.

4. Messages in each queue have equal priorities and they are executed according to their time stamps. A node is capable of processing one message at a time, and when it starts executing a message it will continue until it finishes the execution without being interrupted by another message.

## 4.3 Simulator Basic Structure

The simulator is composed of four main modules, which are represented in the block diagram of Figure 4.2. Each module is implemented in a hierarchal structure of procedures. In this section a general statement about the function performed by each module is introduced, while in Appendix A, structure charts of these modules and definitions of the operational tasks of most of their procedures are presented.

(i) Input Module

This module reads in the network specification provided by the user and expressed as the adjacency list of a testing graph. It converts the information of this list into a linked list of node records and uses the information to initialize some of the fields of each node record. The module also reads in the fault specification of the system.

(ii)Initialization Module

This module serves as a pre-simulation stage. Its function is to initialize the system as well as to prepare some additional data concerning the control of the simulation process and the output representation.

(iii)Simulation Module

As with any simulation software this part is the most important and complex. The simulation algorithm described in the last section is implemented in this module, as well as the diagnosis algorithm.

(iv)Output Module

Output information is collected in this module. This includes a detailed listing of all enqueued messages throughout the simulation, and a summary, which contains the number of produced messages, diagnosis time, and the distribution of all types of messages among the queues at any time. Data for plotting the distribution of messages can also be generated in this module.

The flow of data between the four modules is shown in Figure 4.3. They work in the same sequence as listed above.

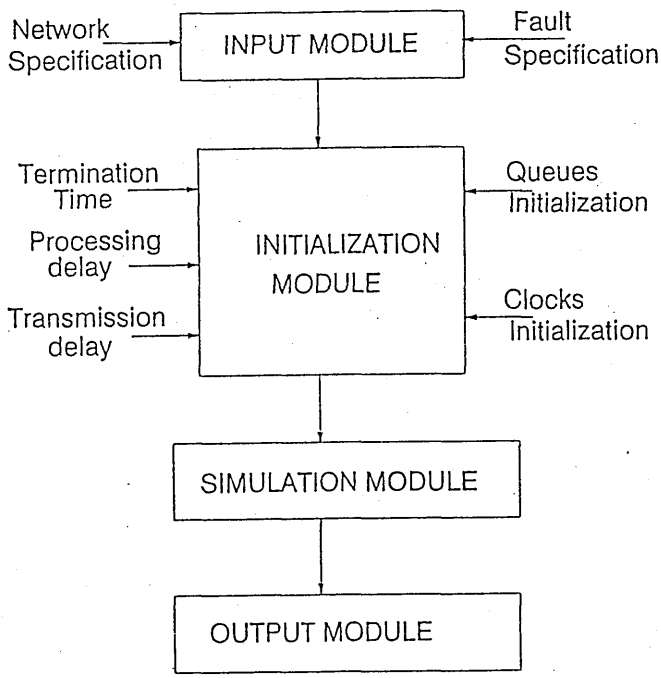



Figure 4.2: The software basic structure

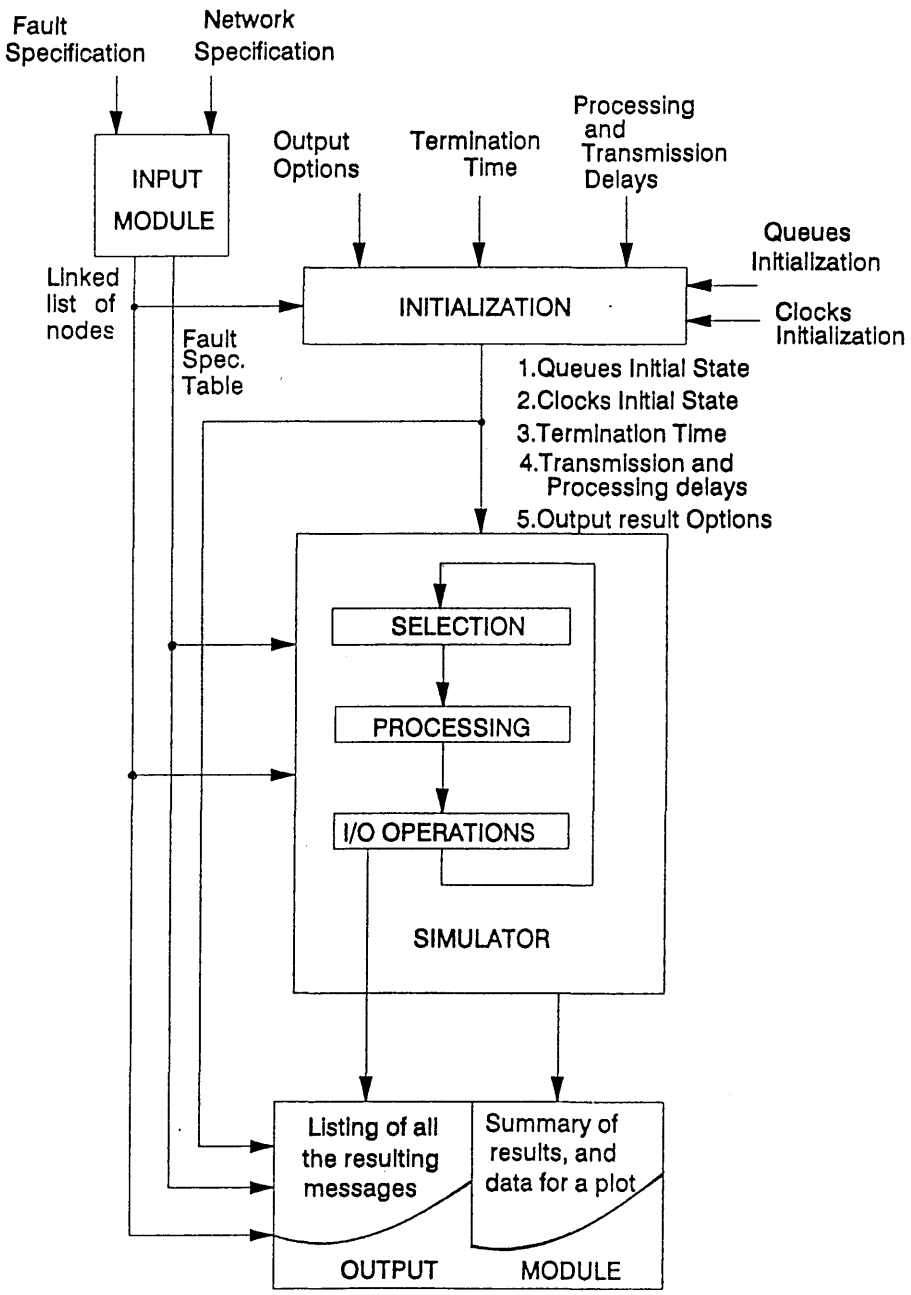



Figure 4.3: The simulator data flow diagram

## 4.4 Data Structure Representation

A dynamic data structure is considered in the software, where a linked list using pointers has been set up. This form of structure is convenient for a system that is composed of an arbitrary number of processing nodes, where it is unnecessary to specify the number of nodes in the linked list [Pete86,Skil84]. The other interesting feature is the possibility of insertions and deletions of list nodes, and updating lists by finding and changing one or more fields of a list node. In the first part of this section it will be shown how this structure is implemented in the program, while in the second part, the way faults are injected in the system will be presented.

### 4.4.1 Network Specification

Network specification are accepted by the program in a tabular form, which represents the testing graph of the system. The table is either given by the user or read from a file. Figure 4.4 shows a sample system, whose network specification are shown in Table 4.1.

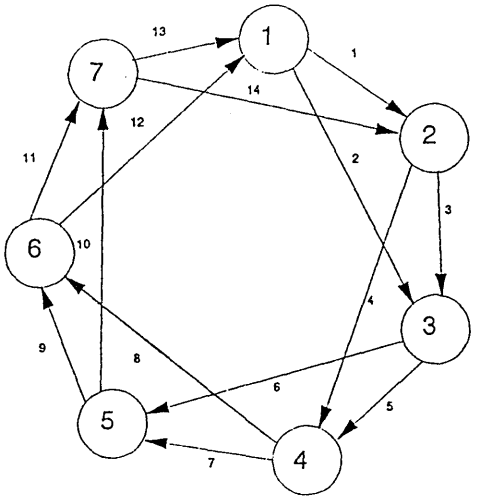



Figure 4.4: Testing Graph for A Sample System

| | | |
|---|---|---|
| 7 | | |
| 14 | | |
| 1 | 1 | 2 |
| 2 | 1 | 3 |
| 3 | 2 | 3 |
| 4 | 2 | 4 |
| 5 | 3 | 4 |
| 6 | 3 | 5 |
| 7 | 4 | 5 |
| 8 | 4 | 6 |
| 9 | 5 | 6 |
| 10 | 5 | 7 |
| 11 | 6 | 7 |
| 12 | 6 | 1 |
| 13 | 7 | 1 |
| 14 | 7 | 2 |

Table 4.1 : Network specification of the sample system of Figure 4.4.

The first and second rows of Table 4.1 contains the number of nodes and the number of links respectively. In the following rows, link numbers are shown at the left followed by the addresses of the nodes they connect. The nodes appear according to their test assignment in the testing graph as testers and testees.

This input data is then processed during the execution of the input module, and hence a linked list is formed with a number of nodes equal to the nodes in the system plus an additional node that is separated from the list, which represents the bus. The function of the last node is to accept messages from all other nodes and pass them to their destinations. Each node in the list is defined as a record containing a number of fields.

The Node Record

All node records are of the same format, and their fields are shown in Figure 4.5. In the program, these fields are defined as follows:

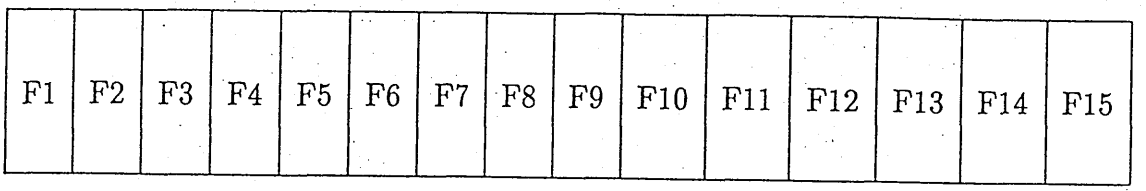



Figure 4.5: Fields of the node record

- F1 (address): This is a sequence number, which is used to identify the node.
- F2 (clock): This is a counter, whose value represents the simulated time of the node.
- F3 (queue): This is a two dimensional array, which is used by the node for storing its own messages.
- F4 (qcount): This specifies the current length of the queue (i.e., the number of messages stored in the queue) at a certain time.
- F5 (outlinks): This is a two dimensional array containing the testee nodes and the links connecting them with the node of this record.
- F6 (inplinks): As in F5, but the nodes contained in this array are the nodes which can test the node of this record as well as the links connecting them.
- F7 (ndfary): This is a one dimensional array, contains a list of the faulty nodes in the system.
- F8 (lnkfary): This is also a one dimensional array. It contains a list of the faulty links that connects a node with its testees.
- F9 (ndset): This is a one dimensional array used to contain the set T, when the node executes an interrogation message.

- **F10 (query-sent)**: This is a one dimensional array. Its elements contain the number of interrogation messages that a node has sent about a specific fault situation and has not received their replies yet.

- **F11 (query-received)**: This is also a one dimensional array. Elements of this array are specified to contain the number of interrogation messages, that a node has received about a specific fault situation.

- **F12 (qcontent)**: This is a two dimensional array used to keep a record of all messages, that are enqueued at a node. The record includes types of messages and their time stamps.

- **F13 (msg-count)**: The value of this field represents the number of messages (diagnosis and others) enqueued at the queue of a node.

- **F14 (diag-msg-count)**: As in F13, but only diagnosis messages are considered.

- **F15 (next)**: A link pointer used to point to the next node in the linked list except for the last node, which is given *nil* reference.

Pointers to link all the nodes, and a pointer to the first node, are used for building a linked list. They have been found very useful for conducting a dynamic search through the list and for updating any field in a node record. Pointers to each individual node, including the one representing the bus are also defined. The linked list representing the sample system of Figure 4.4 is shown in Figure 4.6.

Some of the fields in each node record are initialized during the execution of the input module, while others such as the queues and the clocks, are initialized during the execution of the initialization module as will be shown in the next section.

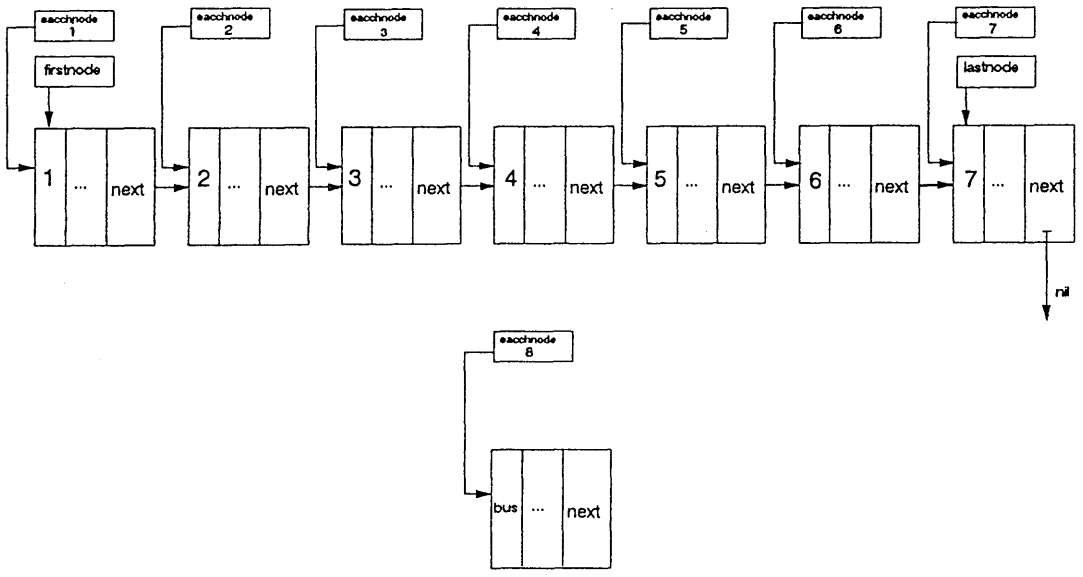



Figure 4.6: A linked list formed for a sample system comprising 7 nodes

### 4.4.2 Fault Specification

This input data is also given to the program in a tabular form. The data in the table represents simulated faults of system elements (both nodes and communication links). To formulate a fault in a node, the address of this node and the time when it has failed will appear in the table. However, links are untested elements in the network, and therefore they cannot be simulated in the same way as for nodes. Modified algorithm SELF3 considers permanent faults only, and when a node is accused of being faulty due to a temporary fault, which is then repaired before the accusation is confirmed by another tester, the initial tester will interpret the situation as a link failure. This feature provides an indirect way for simulating link failures.

If a certain link failure is to be simulated then a temporary fault is assumed at the node, for which this link is used in conducting tests.

A table, which contains a set of simulated faults for the sample system presented earlier is shown in Table 4.2.

| | | | |
|---|---|---|---|
| 2 | 50 | 0 | 0 |
| 5 | 110 | 1 | 115 |
| 999 | 999 | 999 | 999 |

Table 4.2 : Simulated Fault Specification

In Table 4.2 the last row is a sentinel, which indicates the end of the input data. Addresses of faulty nodes and the time when they have failed are in the first and second columns respectively. If the fault is assumed to have been repaired then its corresponding value in the third column is one and the time when it was repaired is in the fourth column. This will simulate a link failure. Otherwise, the fault is permanent and hence its corresponding value in the third and fourth columns is zero.

## 4.5 Pre-Simulation Components

Prior to the execution of the simulation module a number of components should be processed, and some information that will be required in the simulation process should be prepared. Preparation of these components is carried out during the execution of the initialization module, and this section will be devoted to describing them.

### 4.5.1 Queues Initialization

One of the assumptions on which the simulator is based is that when all queues are empty or having no messages to be processed, the simulator will terminate. Therefore it is necessary to specify some initial conditions for the queues. Queues initialization involves placing a message (or more) onto at least one queue. The program offers two ways for doing this, either randomly or from a file. Random initialization results in generating a random number of messages, which are of random type, and distributed randomly among the queues. It is quite possible that none of the generating messages is useful in simulating a specific case. However, they can still be used to provide some information about the general performance of the program.

Alternatively, queues can be initialized from a file, which can either be prepared before, or given at the time when the program is executing the initialization module. In this case, messages are normally chosen to match fault specification, which should have been provided earlier. The basic principle of this match is that when a fault is simulated in a node at some time, the node should be tested after an appropriate time in order to detect this fault and hence diagnose it.

For the sample system presented in Figure 4.4, one of the possible patterns for initializing the queues is shown in Table 4.3.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | | | | | | | |
| 111 | 999 | 999 | 1 | 2 | 0 | 0 | 55 |
| 0 | | | | | | | |
| 0 | | | | | | | |
| 1 | | | | | | | |
| 111 | 999 | 999 | 4 | 5 | 0 | 0 | 110 |
| 0 | | | | | | | |
| 0 | | | | | | | |
| 0 | | | | | | | |

Table 4.3 : A Possible initialization pattern for the queues of the sample system of Figure 4.4.

Messages in the above table are placed in the queues of nodes 1 and 4, while the rest of the queues are left empty. This complies with the pattern of simulated faults in nodes 2

and 5, shown in Table 4.2. Both messages are of type TEST, and they form part of the local testing schedule of the two nodes.

Any message that appears in a queue should be of the format that was shown in Figure 4.1. The notation appeared in that Figure will be used to describe the initialization messages of Table 4.3. For instance, the number 111, which corresponds to the information of the field A1, will be translated by the program as a message of type TEST, and if the position A4 contains the address of the initial tester then positions A2 and A3 will not be considered. Instead, they will be assigned some distinguishable values (assumed here as 999). This case appears in the test messages that forms part of a local testing schedule of a node. On the other hand, if the test message is generated in response to an interrogation, then positions A2 and A3 will contain the initial tester and the accused node respectively. The function of the remaining elements in the previous test messages are similar to what was described earlier in section 4.2.3.

As for other types of messages, it is important to notice that in the program their types are also assigned to numbers similar to the test message, where NDAC = 222, QUES = 333, PASS = 444, FAIL = 555, NDFL = 666, and INFO = 777. In the message of type NDAC, positions A4 and A5 are not used, since this message is destined to the queue of the node itself.

### 4.5.2 Initialization of Local Clocks

It was mentioned earlier that each node in the simulated system is provided with a local clock, whose value represents the simulated time of the node. Clocks are automatically initialized with zeros during the execution of the input module. However, at this stage, the user will be given a choice for re-initialization, and if this choice is selected then the new values can be keyed in.

### 4.5.3 Termination Time

The simulation is assumed to continue as long as the simulated system is still capable of communicating, and has messages to be processed. The risk of queue overflow is reduced by continuously removing old messages from them. However, for a long simulation, clocks cannot count indefinitely. Thus, a bound of a relatively large value is defined, which represents the simulator termination time. Each time a message is executed the value of the clock of the selected node is checked and if the termination time is being

reached then the simulation will terminate, regardless of the status of the queues.

The user will be given a choice at this stage to change the value of termination time, if required.

### 4.5.4 Delays in The System

It is assumed that there exist two main sources of delay in the system. A delay during the processing of a message and a delay during its transmission. Some of the factors affecting the time required for message processing are the processor speed, the length of its queue, and the actions required to execute a certain type of message. As for transmission delay, it is mainly the effect of the communication network, that is considered.

If a node is to put a message onto its own queue, then there will be no transmission delay. However, messages that are directed to a remote destination will require some time for transmission. The topology of the communication network, the way it has been utilized, and the length of the message are some of the factors on which transmission delay is related.

Both processing and transmission delays are assumed to be deterministic, and have been given fixed values, in the program. These values are prepared before starting the simulation, and the user is given a choice for altering them.

### 4.5.5 Output Results Options

After each run, the program will always produce an output file in which a summary of the run is written. However, before the simulation process starts, the user is introduced by an output options menu, where options can be entered. The selected options will be used to produce additional output files during the execution of the output module.

## 4.6 Basic Characteristics of The Software

Some of the basic characteristics of the software are listed below:

(i) The Modular Design Approach

This design approach is implemented, and the package has been divided into four main modules with each one being assigned a distinct function. The control to these modules is carried out by the main program. Each module is divided into several sub-modules to perform its detailed needs. There are some procedures that are common to several sub-modules and few to more than one module. This form of design makes it easier for

the package to be understood and modified.

(ii) Interactive Operating Mode

This operating mode has been used to run the software on the host computer. The user, who does not need a previous knowledge about the internal structure of the package can easily control its operation, where clear and various ways to input the data or output the results are being provided. Network specification, fault specification, initialization of the queues, the clocks initial state, time delays, termination time, and the form of output required can all be given and modified by the user during the running of the program. This feature allows for the simulation of different systems with various assumptions without having to change the internal structure of the program.

(iii) Error Handling Facility

The package is being provided by an error handling capability, so that if an incorrect input data is mistakenly entered, the user will either be warned by an appropriate message and prompted again or only prompted without a warning message to re-enter the correct data.

(iv) Asynchronous Event Driven Technique

Among many techniques that can be used in simulation, this one has been incorporated in the implementation of the software. It reflects the way in which a distributed multicomputer system is working, and in terms of simulation it reduces the run time and hence increases the program efficiency.

(v) Programming Language

PASCAL high-level language has been used for writing the software. As well as being a structured language, PASCAL allows the use of linked lists. This has been very useful in describing the system nodes as records and fields. The use of pointers makes it easy to search through the node records and to perform processes on their fields.

The software package is composed of approximatley 4200 PASCAL language instructions.

(vi)Input and Output Files

The input data to the package may be held by a read-only file, which can be prepared before running the program, while a number of output files are opened to write the output.

In a distributed system, the lack of global memory(queue) to provide a global state of the system makes the investigation and hence verification of distributed algorithms tedious and inclined to errors. Moreover, it makes the representation of output results difficult. To overcome this implication, a snapshot of the messages contained in all distributed queues is taken and written into an output file, after each message execution. Each snapshot represents the condition of the system at the time when it is taken. The information collected from all the snapshots will be used to investigate the behaviour of the distributed algorithm and the evolution of the system condition in time.

In other files, only selected information of each enqueued message are written. They include the message type and its time stamp. The information collected in these files are used to summarize the output and show the distribution of messages among the queues in time, which can then be represented as a plot.

(viii) Removing Unnecessary Data

The package is provided with a facility for cleaning up the queues from old messages that has been executed and no longer needed. This scheme is applied periodically and it minimizes the risk of queues overflow and ensures that newly generated messages can be enqueued.

# Chapter 5

# Simulation Results

## 5.1 Introduction

In this chapter, the results are given for the simulation of a number of systems. Two types of networks are considered, the H(n,r;$k_1, ...k_r$) graph networks and the 4-dimensional hypercube. Different fault situations are assumed and their diagnosis procedures are illustrated. Two terms relating to diagnosis(number of diagnosis messages and diagnosis time) were defined in chapter 4. When a specific fault situation is to be diagnosed then the reduction of either the number of produced messages or the diagnosis time or both, while preserving the efficiency of diagnosis is of obvious advantage. An analytical study of this issue is included in this chapter.

The 'diagnosis time' was defined as the time required for the network to complete the diagnosis of a specific fault situation. The complete diagnosis is assumed to consist of four phases; Testing and accusation, Interrogation, Reply and Broadcasting phases. The four diagnosis phases are defined as follows:

*Definition 5.1: Testing and accusation phase.* In this phase a test is applied by a tester on one of its testees. If there is a fail test result then the testee is accused by its tester of being faulty. The tester and testee nodes, which are included in this phase are, sometimes called the accuser and accused nodes respectively.

*Definition 5.2: Interrogation phase.* This phase follows the testing and accusation and it includes the interrogation messages(messages of type QUES), which are sent regarding the condition of the accused node. The phase is initiated by the accuser and finishes either by finding another fault-free tester of the accused node or by failing to find the required tester, in which case the diagnosability of the system is exceeded.

*Definition 5.3: Reply phase.* If a tester is found in the interrogation phase it will test the accused node and in the reply phase, the test result will be sent back following the paths,

which were traversed by the interrogation messages during the interrogation phase in an opposite direction .

*Definition 5.4: Broadcasting phase.* This phase starts after the accuser node receives a reply, which confirms its accusation(i.e., the accused node is found faulty). The phase includes sending messages to inform the fault-free nodes about the failure of the node that has just been diagnosed. It is not required for the diagnosis to go through this phase, in case of a link failure, since only the local data base of the accuser node need to be updated.

A path that is traversed by one or more interrogation messages during the interrogation phase, while searching for a tester is referred to as the *interrogation path.*

## 5.2 Examples

The graph networks, which are generated using the H-graphs were introduced in section 2.3.2. The H(7,2;1,2) was used in chapter 4 as a sample system to illustrate the formation of the data structure of the simulator. In this section, this same graph network is used to demonstrate the use of the simulator and to show the diagnosis procedure of some simulated faults. While describing the construction method of these networks in section 2.3.2, a labeling of *(0 - (n-1))* was used to identify the nodes. In the following examples as well as in the sample system used in chapter 4, the nodes are addressed (*1 - n*). This way of addressing is found more convenient when applying the simulator. Another assumption that is common to all the examples which will follow, relates to the way the diagnosis time is being evaluated. Fixed values are assumed for the processing and transmission delays of every message. These values are; *processing time := 2 units,* and *transmission time := 1 unit.*

### 5.2.1 The H(7,2;1,2) graph network

**Example 5.1 :** The H(7,2;1,2) graph network is shown in Figure 5.1. For this network, consider part of the fault specification shown in Table 4.2. Let node 2 fail at time 50 and not be repaired. Assume that at this or at a later time, say 55, this node has been tested by node 1, where node 1 is assumed fault-free. Node 1 will receive a reply of type FAIL for its test and as far as it is concerned, this test result could be either due to a failure in node 2 or in the link connecting node 1 and node 2. Consequently, node 1 will accuse node 2 of being faulty and start a diagnosis procedure, which will involve other

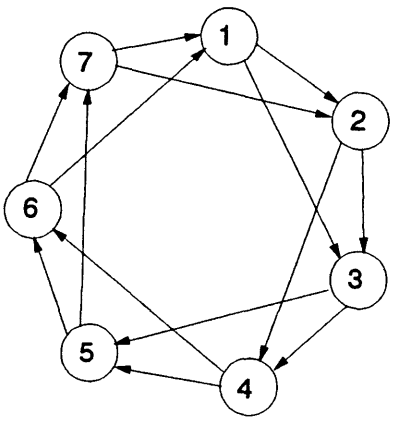



Figure 5.1: The H(7,2;1,2) graph network

fault-free nodes in the network trying to find another tester of node 2. The diagnosis procedure will pass through the four phases, defined earlier, as shown in Figure 5.2.a-d. In this and next similar Figures, a message that is being transmitted through a test link is expressed by its type and time stamp using this form; *message type/time stamp.*

In Figure 5.2.a, the testing and accusation phase, started with node 1 applying a test on node 2. The test has failed, therefore the phase finishes with node 1 accusing node 2 of being faulty. In Figure 5.2.b, being an accuser, node 1 will start the next phase of the diagnosis by interrogating node 3 regarding the condition of node 2. Node 3 then interrogates node 4 and node 5, which means that the interrogation messages are starting to traverse two interrogation paths. It should be noticed that interrogation paths must be kept acyclic, otherwise the diagnosis will run into a great difficulty as will be shown in the next section. To prevent a path from being cyclic, a set of nodes(set T) are included in all interrogation messages and hence any node must not appear more than once in this set. From node 5, two interrogation messages were generated and directed to nodes 6 and 7, where both nodes are not included in set T, which is contained in the message QUES/65 received at node 5. At node 6, however, two interrogation paths are converged at the same time. This node allows the continuation of one of them, while replying to the other with a message of type FAIL (FAIL/73 to node 5). In fact node 6

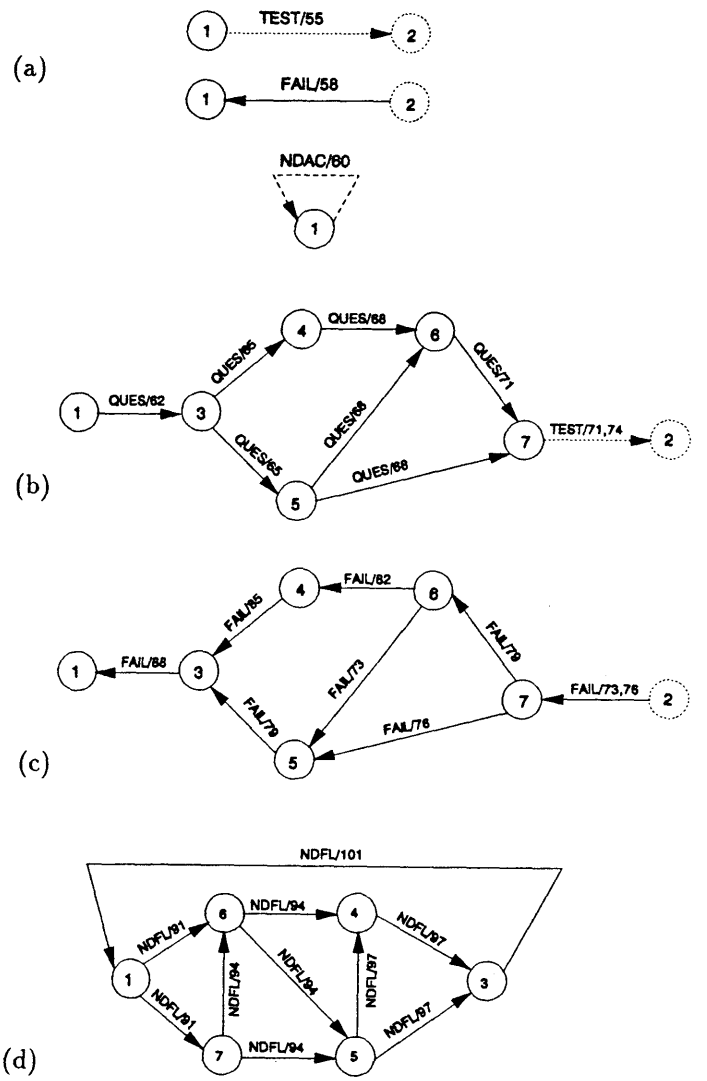



Figure 5.2: Diagnosis phases for a simulated node failure in the H(7,2;1,2) graph network. (a):Testing and accusation phase, (b):Interrogation phase, (c): Reply phase, (d): Broadcasting phase.

has two other alternatives regarding the execution of the messages received through these two paths. All three alternatives will be studied in the next section. The interrogation phase finishes by finding a fault-free tester(node 7) at which two interrogation paths have converged at two different times. Node 7, being a tester, is assumed to deal with every interrogation message independently, and therefore executes all messages and replies to them asynchronously as shown in Figure 5.2.c. After applying a test, node 7, will find node 2 faulty, therefore a message of type FAIL will be sent back during the reply phase (Figure 5.2.c). Node 5, which has sent two interrogation messages did not pass the reply it has received from node 6 at 73 (FAIL/73). Instead, it has waited until the reply from node 7 (FAIL/76) is being received. Similarly, node 3 has waited for the reply from node 4 before sending the test result to node 1. At node 1, since the test result confirms the accusation, the broadcasting phase will start with this node updating its list of faulty nodes by including node 2 and sending a message of type NDFL to its fault-free testers. A node which receives this message for the first time will update its list of faulty nodes and pass the message to its fault-free testers. Eventually, a copy of this message will reach all fault-free nodes in the testing graph. No message informing of a specific faulty node is allowed to pass through a fault-free node more than once. At the end of this phase, the faulty node is considered to be disconnected from the system, and the system will continue working in a degraded mode.

A plot showing the messages produced by the nodes of the system for diagnosing the condition of node 2 versus the simulated time is shown in Figure 5.3. A total of 58 diagnostic messages are produced and the diagnosis time required to complete the diagnosis is 40.

**Example 5.2 :** The H(7,2;1,2) graph network is $1_{n,l}$-fault self-diagnosable according to Theorem 3.1. However, it is possible for the diagnosability of this system to be more than this value if the faults occurring are not consecutive. This example will demonstrate this facility as well as the diagnosis procedure of a link failure. Consider Example 5.1, and assume that the accused node (node2) was not faulty but the link connecting node 1 and node 2 is faulty. In the diagnosis of this fault, the test and accusation and the interrogation phases are exactly the same as in Figure 5.2.a,b. The reply phase, however, is different, where a message of type PASS will be routed during this phase. Once this message is received at node 1, only the list of faulty links of this node are updated by

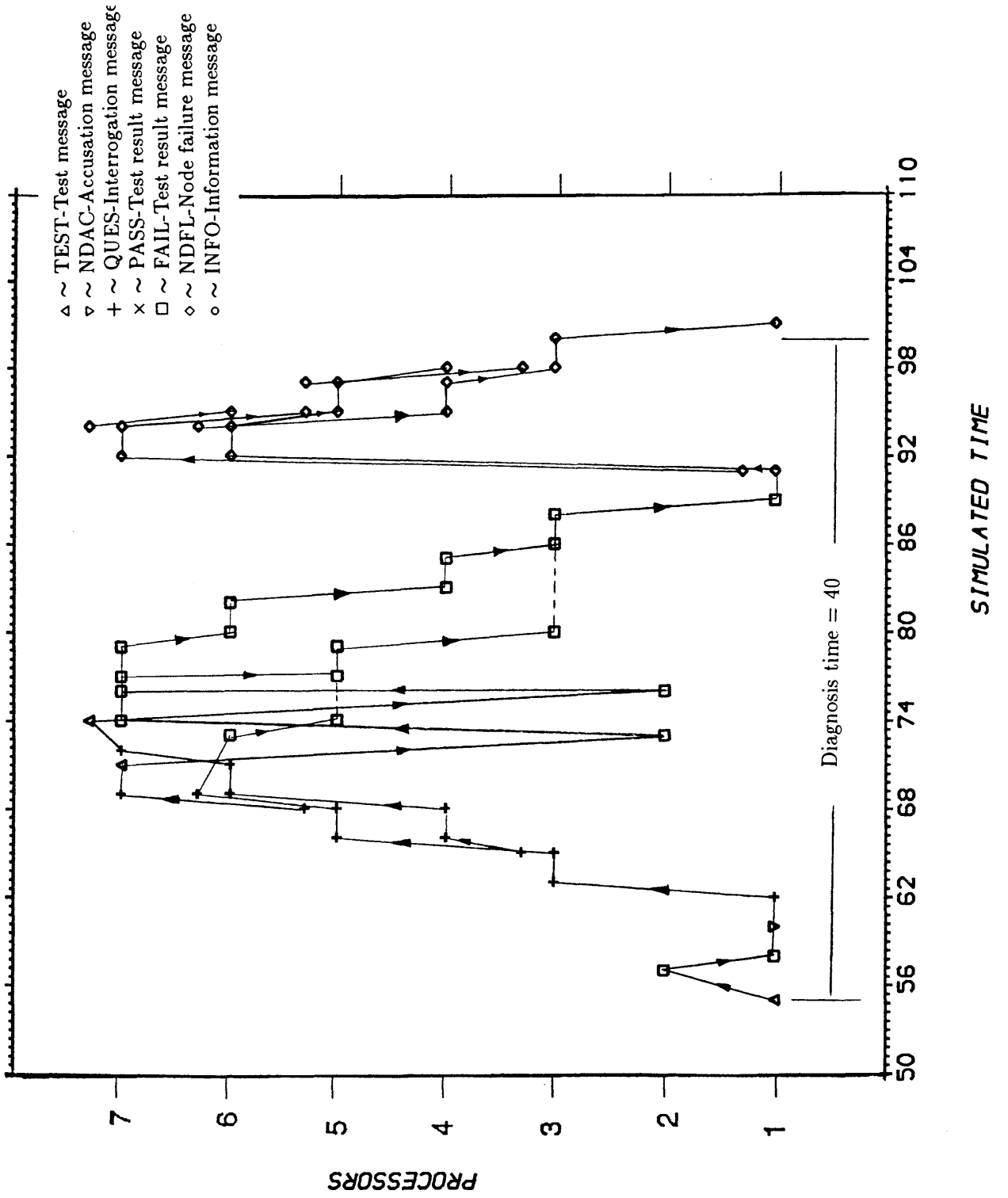



Figure 5.3: Messages produced vs. simulated time during the diagnosis of a simulated node failure in the H(7,2;1,2) graph network.

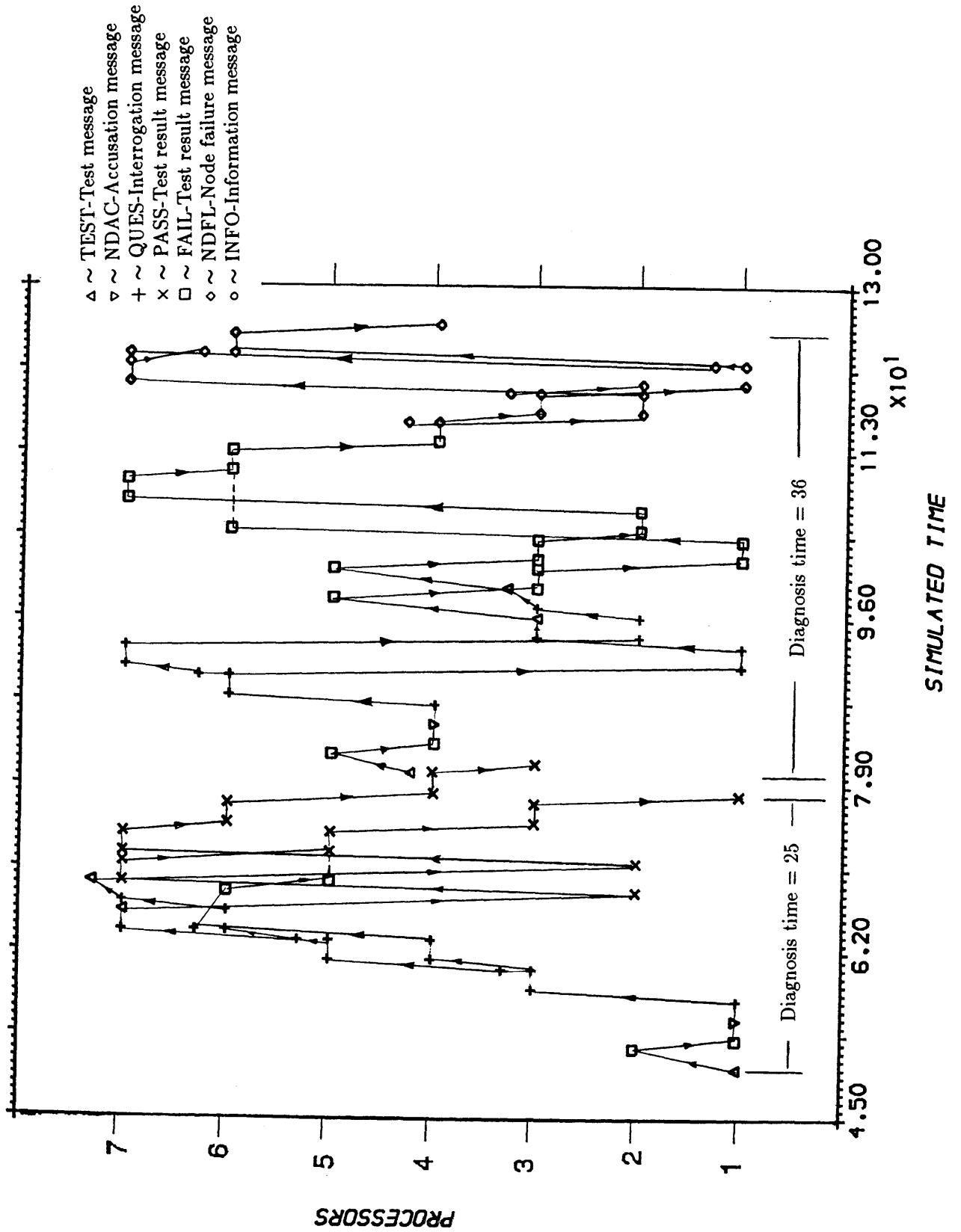



Figure 5.4: Messages produced vs. simulated time during the diagnosis of simulated link and node failure in the H(7,2;1,2) graph network.

including the link connecting it with node 2. Hence, the diagnosis will be finished without going through the broadcasting phase. After disconnecting the link between node 1 and node 2, it will not be possible to diagnose the condition of node 2 if a test on this node fails. However, it will be possible to diagnose the condition of other nodes in the system if they are accused of being faulty. Assume therefore that a test on node 5 by node 4 has failed and hence node 4 has accused node 5 of being faulty and started the diagnosis procedure. The messages that are generated during the diagnosis of both this simulated fault and the link failure are shown in Figure 5.4, and from this Figure it can be noticed that there are no messages transmitted or received between nodes 1 and 2 because the link connecting them was diagnosed as faulty and is therefore not used.

**Example 5.3 :** Reconsider the H(7,2;1,2) graph network, and assume that the system is being assigned a computational task, which is represented by a message of type INFO, at time 0. The message, which has no role in the diagnosis, will travel between the nodes following fault-free paths. Assume now that the fault situation in Example 5.1 has occurred, where a fault in node 2 is being diagnosed. In Figure 5.5, the messages that were produced in the network before the occurrence of the fault are all of type INFO, and related to the simulated computational task. After diagnosing the fault, and hence isolating node 2 from the system, these messages are still travelling in the network, however, the path passing through node 2 is being avoided.

Note that because the system is involved in a simulated computational task, the diagnosis time is slightly higher than that in Example 5.1, i.e., 42 units compared to 40 units in Example 5.1 are required.

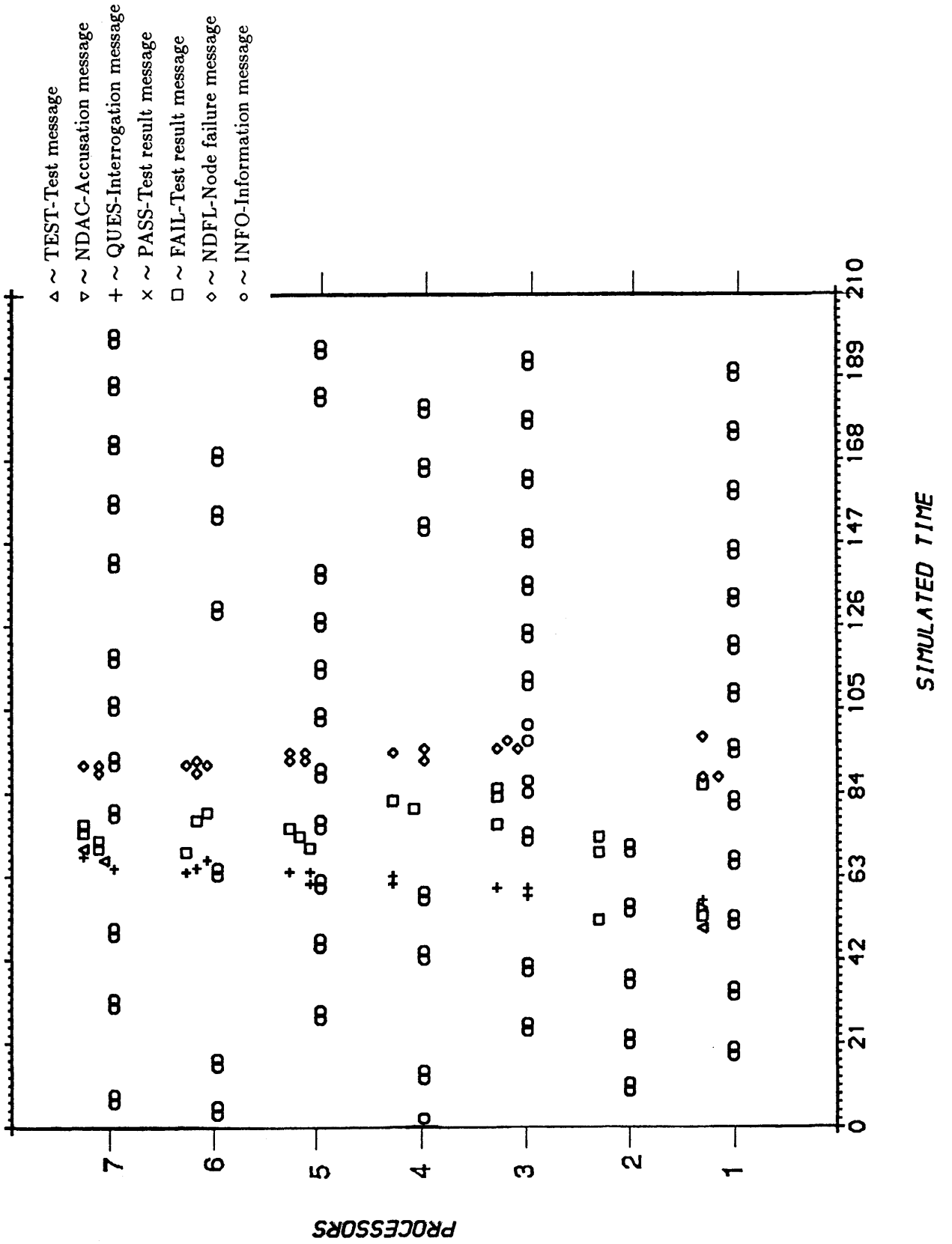



Figure 5.5: Messages produced vs. simulated time for a simulated computational task and the diagnosis of a node failure in the H(7,2;1,2) graph network.

### 5.2.2 A Hypercube Architecture

The hypercube is a well-known example of a highly structured communication architecture. We will consider the structure of a binary cube for which the number of nodes $n$ is always a power of 2 and for $n = 2^m$, the nodes can be expressed 0 through $(n-1)$ with $m$-bit binary numbers. In the following examples, however, we will address the nodes 1 through $n$ and consider only the testing graph of these networks. A four-dimensional hypercube, which has $2^4$ nodes is shown in Figure 5.6 and in this Figure, each node is assigned to test, and be tested by, two other nodes.

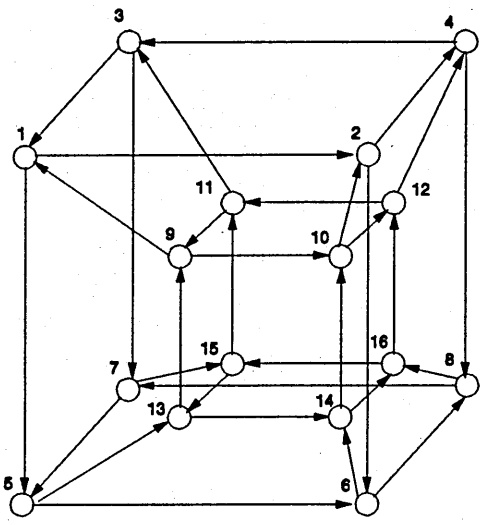
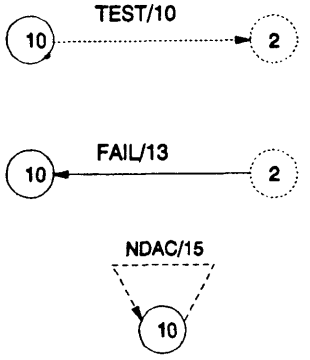
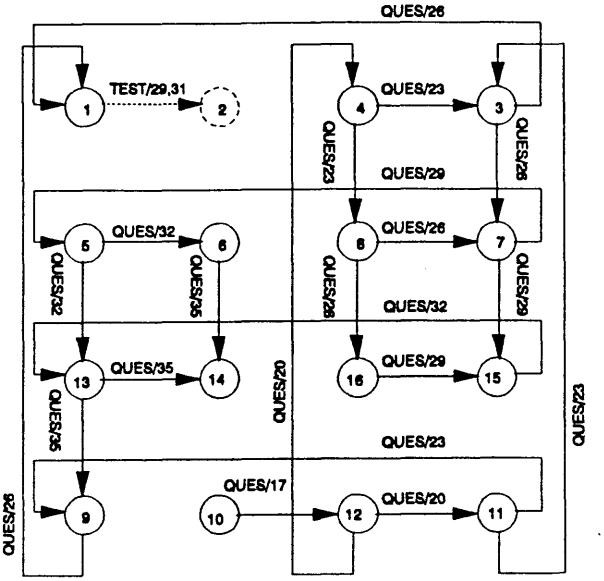



Figure 5.6: Testing graph for a four-dimensional hypercube

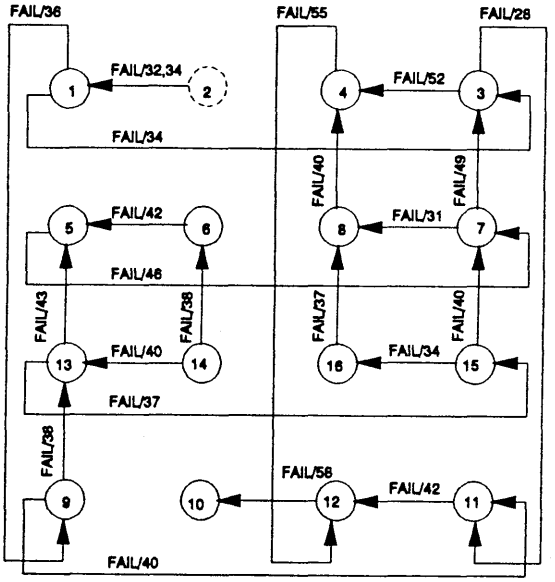
Example 5.4 : Consider the 4-dimensional hypercube shown in Figure 5.6 and let a test on node 2 by node 10 fail at time 13 as shown in Figure 5.7.a. Node 10 requires the help of the rest of the system in order to diagnose the actual condition of node 2, therefore a diagnosis procedure started at this node. In a similar way to what was described in Example 5.1 the diagnosis procedure will go through the four phases shown in Figure 5.7.a-d. From this it can be noticed, that after the end of the reply phase, node 2 was found faulty and therefore its diagnosis was continued by broadcasting its condition to all fault-free nodes in the network. Figure 5.8 shows a plot of diagnosis messages versus simulated time as well as the paths traversed by messages throughout the nodes.



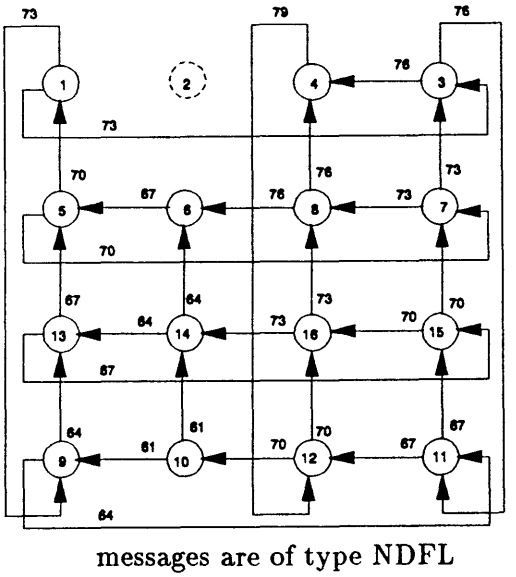

(a)




(b)




(c)




(d)

Figure 5.7: Diagnosis phases of a simulated node failure in the four-dimensional hypercube. (a) Testing and accusation. (b) Interrogation. (c) Reply. (d) Broadcasting.

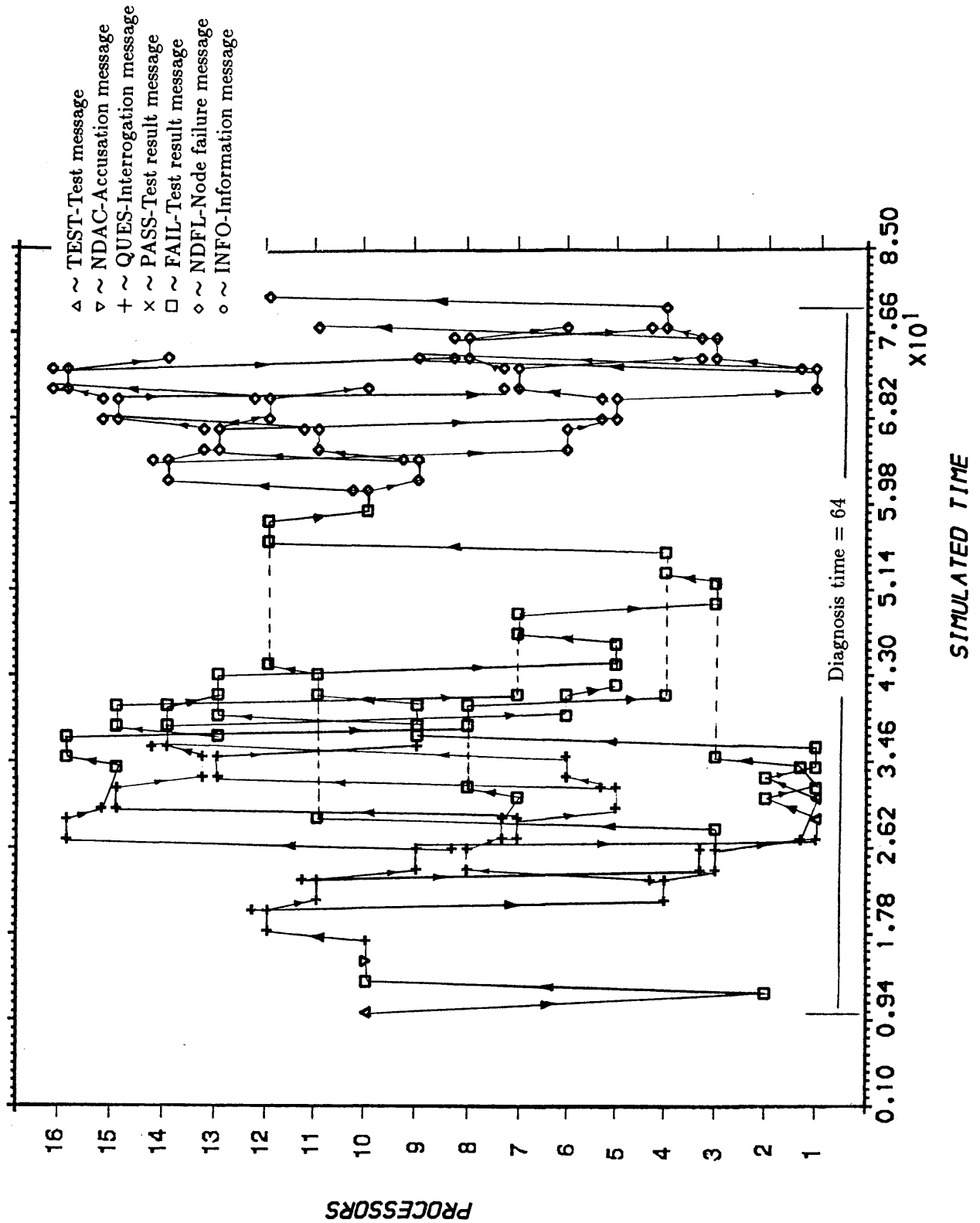



Figure 5.8: Messages produced vs. simulated time during the diagnosis of a simulated node failure in the 4-dimensional hypercube.

**Example 5.5 :** Consider the 4-dimensional hypercube considered in the last Example and assume that after diagnosing the fault in node 2 and hence isolating this faulty node, another fault was detected, when a test on node 15 by node 16 has failed causing the accusation of node 15. In a similar diagnosis procedure, node 15 was completely diagnosed as faulty. A plot of the messages produced versus simulated time during the diagnosis of both node failures is shown in Figure 5.9.

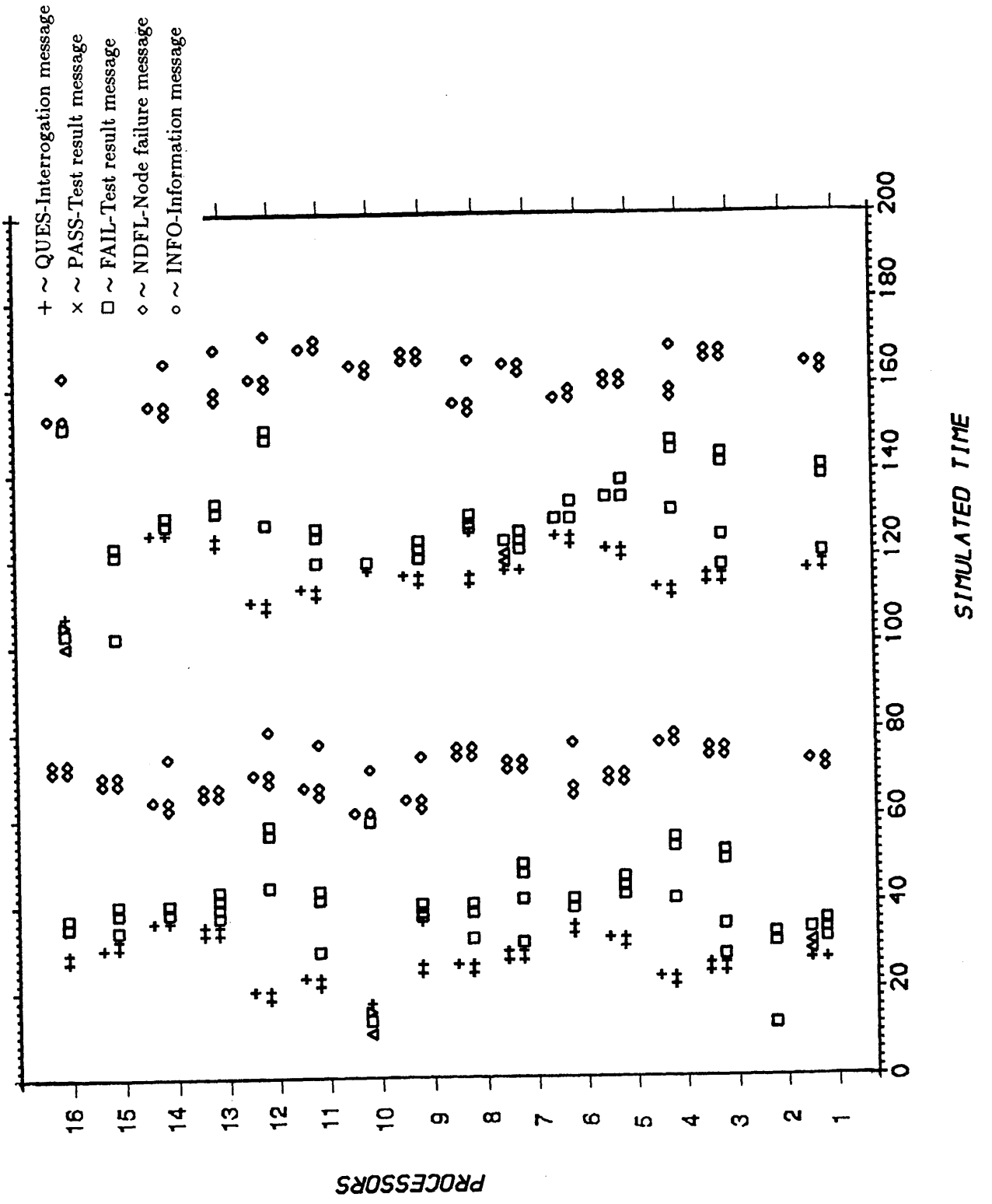



Figure 5.9: Messages produced vs. simulated time during the diagnosis of two simulated node failures in the 4-dimensional hypercube

## 5.3 Prevention of Redundant Parallel Paths

### 5.3.1 Introduction

It was mentioned in this and previous chapters, while describing modified algorithm SELF3 [Hoss88], that every interrogation message includes a set of nodes called *set T*, which is used to ensure that interrogation messages travels only through acyclic paths. At a node $P_q$, which has accused another node $P_r$, the interrogation message(s) regarding the condition of $P_r$, that is(are) sent by $P_q$ will have a set T, which is defined by; $T =$ *all fault-free testees of* $P_q$. Consider that one of these messages is being received by node $P_m$, which is not a tester of node $P_r$. Node $P_m$ will interrogate its fault-free testees $FT(P_m)$, provided that they are not included in set T of the message it has received. The interrogation messages, which node $P_m$ will send, are provided with a set T, that is modified into $T = T \bigcup FT(P_m)$.

While the interrogation paths are branching in their search for a tester of the accused node, it is possible for a single node, especially in a graph with long paths, to be involved with more than one interrogation path. We will continue with our proposed case in which $P_q$ accused $P_r$ and assume that node $P_t$, which is not a tester of node $P_r$, has received messages from $P_m$ and $P_n$ interrogating about the condition of $P_r$. Let these carry sets $T_m$ and $T_n$ respectively. If this case is to be simulated, various situations can arise due to the differences in sets $T_m$ and $T_n$ and to the sequencing of execution of the messages by node $P_t$. In this section, these situations will be investigated.

Let the fault-free testees of $P_t$ be $FT(P_t)$ then these testees should be related to the sets $T_m$ and $T_n$ according to one of the following possibilities;

(a) $FT(P_t) \bigcap \overline{T_m} = FT(P_t) \bigcap \overline{T_n}$

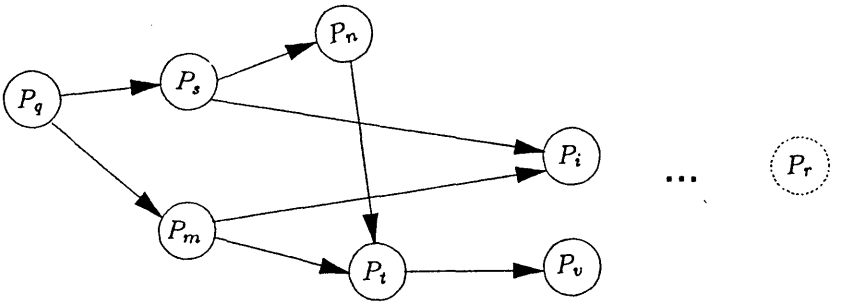
(b) $FT(P_t) \bigcap \overline{T_m} \supset FT(P_t) \bigcap \overline{T_n}$

(c) $FT(P_t) \bigcap \overline{T_m} \subset FT(P_t) \bigcap \overline{T_n}$

(d) $FT(P_t) \bigcap \overline{T_m} \not\supset FT(P_t) \bigcap \overline{T_n}$ and $FT(P_t) \bigcap \overline{T_m} \not\subset FT(P_t) \bigcap \overline{T_n}$

(e) $FT(P_t) \bigcap \overline{T_m} = \phi$ and $FT(P_t) \bigcap \overline{T_n} = \phi$.

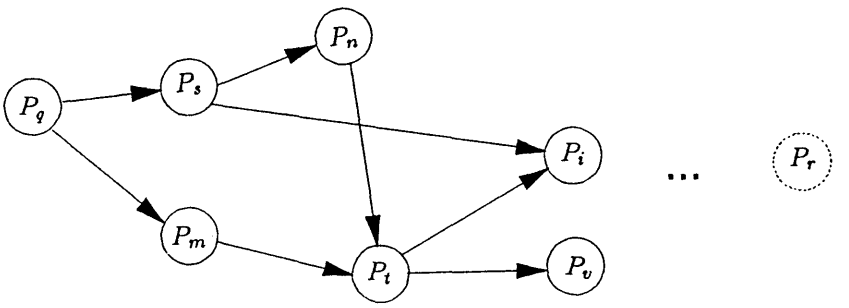
These five possibilities are illustrated in Figure 5.10.a-e. Each case(a-e) is assumed to represent part of an interrogation phase during a diagnosis procedure of node $P_r$. In this Figure, we assume that node $P_t$ originally has a set of fault-free testees composed of,




$T_m = \{P_q, P_s, P_m, P_i, P_t\} \Longrightarrow \mathrm{FT}(P_t) \bigcap \overline{T_m} = \{P_v\}.$

$T_n = \{P_q, P_s, P_m, P_n, P_i, P_t\} \Longrightarrow \mathrm{FT}(P_t) \bigcap \overline{T_n} = \{P_v\}.$

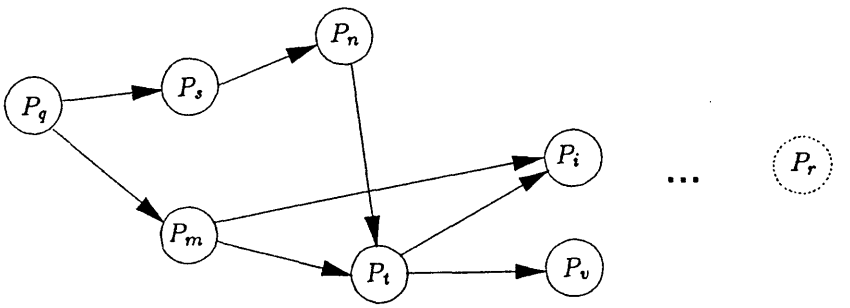
(a) $FT(P_t) \bigcap \overline{T_m} = FT(P_t) \bigcap \overline{T_n}$




$T_m = \{P_q, P_s, P_m, P_t\} \Longrightarrow \mathrm{FT}(P_t) \bigcap \overline{T_m} = \{P_i, P_v\}.$

$T_n = \{P_q, P_s, P_m, P_n, P_i, P_t\} \Longrightarrow \mathrm{FT}(P_t) \bigcap \overline{T_n} = \{P_v\}.$

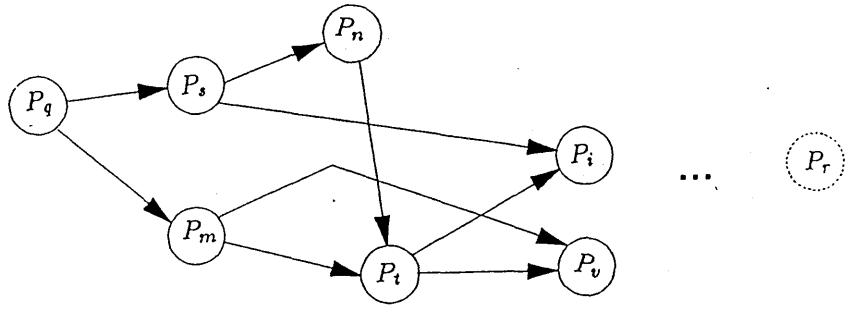
(b) $FT(P_t) \bigcap \overline{T_m} \supset FT(P_t) \bigcap \overline{T_n}$




$T_m = \{P_q, P_s, P_m, P_i, P_t\} \Longrightarrow \mathrm{FT}(P_t) \bigcap \overline{T_m} = \{P_v\}.$

$T_n = \{P_q, P_s, P_m, P_n, P_t\} \Longrightarrow \mathrm{FT}(P_t) \bigcap \overline{T_n} = \{P_i, P_v\}.$

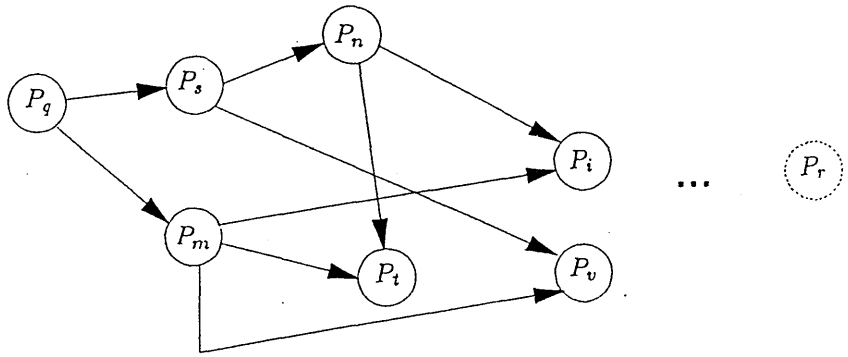
(c) $FT(P_t) \bigcap \overline{T_m} \subset FT(P_t) \bigcap \overline{T_n}$




$T_m = \{P_q, P_s, P_m, P_v, P_t\} \Longrightarrow \mathrm{FT}(P_t) \bigcap \overline{T_m} = \{P_i\}.$

$T_n = \{P_q, P_s, P_m, P_n, P_i, P_t\} \Longrightarrow \mathrm{FT}(P_t) \bigcap \overline{T_n} = \{P_v\}.$

(d) $FT(P_t) \bigcap \overline{T_m} \not\supset FT(P_t) \bigcap \overline{T_n}$ and $FT(P_t) \bigcap \overline{T_m} \not\subset FT(P_t) \bigcap \overline{T_n}$




$T_m = \{P_q, P_s, P_m, P_i, P_v, P_t\} \Longrightarrow \mathrm{FT}(P_t) \bigcap \overline{T_m} = \phi.$

$T_n = \{P_q, P_s, P_m, P_n, P_i, P_v, P_t\} \Longrightarrow \mathrm{FT}(P_t) \bigcap \overline{T_n} = \phi.$

(e) $FT(P_t) \bigcap \overline{T_m} = \phi$ and $FT(P_t) \bigcap \overline{T_n} = \phi.$

Figure 5.10: Different possibilities of $FT(P_t)$ with respect to $T_m$ and $T_n$

$P_i$ and $P_v$ and the nodes in this set are, for each case (i.e., a-e), assumed to be related differently, as testers and testees, to nodes $P_m$ and $P_n$. This difference in relation between nodes $P_m$, $P_n$, $P_i$ and $P_v$ will cause the differences between $FT(P_t) \bigcap \overline{T_m}$ and $FT(P_t) \bigcap \overline{T_n}$.

Assume first that, when node $P_t$ executes the two messages, the actions that will be taken depend only on the contents of the sets $T_m$ and $T_n$ and not on which message is first or last to be executed. The situations for cases (d) and (e) are straightforward and therefore they will only be defined briefly in this section, where in case (d), node $P_t$ will interrogate a different set of testees when it executes the messages from $P_m$ and $P_n$. Node $P_i$ will be interrogated when the message from $P_m$ is executed, while node $P_v$ is interrogated when the message from $P_n$ is executed. In case(e), however, the execution of each one of the two messages by node $P_t$ will generate a reply of type FAIL, since this node is neither a tester of node $P_r$ nor has some fault-free testees that are not included in the sets $T_m$ or $T_n$ to interrogate.

Meanwhile, in cases(a-c), the situation is somehow different. In case(a), for instance, the same set of testees will be interrogated whichever message is executed first, and this set will be interrogated again, when the other message is executed. In (b) and (c), however, the testees which will be interrogated twice, after executing the two messages, are those which occur in both $FT(P_t) \bigcap \overline{T_m}$ and $FT(P_t) \bigcap \overline{T_n}$. By interrogating a set of testees, or part of it, for the same reason more than once, node $P_t$ has in fact repeated similar actions. This has happened in cases(a),(b) and (c), where this node has executed all the interrogation messages it has received independently, considering only the contents of the set T, and this in fact complies with what is recommended by the algorithm of [Hoss88]. In this algorithm, a node like $P_t$ is not required to consider its previous actions before sending an interrogation message to another node. Such a practice will result in generating more interrogation messages and hence the formation of more parallel paths, which can be considered as *redundant*. These extra messages have no advantage to the diagnosis process. On the contrary, they may incur additional delay in performing the diagnosis. We consider that their prevention is of potential advantage and therefore assume that it is important for a node not to interrogate another node more than once for the same reason, when these two conditions hold;

1. the first interrogation message, that has been sent is still waiting for a reply, and

2. the difference between the value of the time stamp of the first message, whose reply has not been received yet, and the current clock value is within a specific timeout period.

The value of the timeout period may vary according to the size of the system and hence the expected length of the interrogation path. According to this assumption and the two conditions included in it, the situation at node $P_t$ will be reassessed and in this assessment, we will assume that conditions (1) and (2) above are always holding (as it is usually the case). Thus, for (a), if the message from $P_m$ is assumed to be executed first then all the testees in $FT(P_t) \bigcap \overline{T_m}$ will be interrogated. At a later instance, however, when node $P_t$ executes the message from $P_n$ none of the nodes in $FT(P_t) \bigcap \overline{T_n}$ need to be interrogated because they have already been interrogated. In contrast, if node $P_t$ has executed the message from $P_n$ first, all the nodes in $FT(P_t) \bigcap \overline{T_n}$ will be interrogated, while none of the nodes in $FT(P_t) \bigcap \overline{T_m}$ need to be interrogated when executing the message from $P_m$ .This is not the case for (b) and (c), however, where the precedence of executing the two messages makes a difference in the actions that node $P_t$ has to take. Consider case (b) and assume that the message from $P_n$ has been executed first, then all the testees $FT(P_t) \bigcap \overline{T_n}$ will be interrogated by $P_t$. Consequently, when the message from $P_m$ is to be executed, only part of $FT(P_t) \bigcap \overline{T_m}$ will be interrogated. This part includes the nodes, which do not exist in $FT(P_t) \bigcap \overline{T_n}$ and hence have not been interrogated. For the same case, (i.e., case(b)) , if the message from $P_m$, is executed first, it will result in interrogating the testees $FT(P_t) \bigcap \overline{T_m}$ by node $P_t$, and because this set includes $FT(P_t) \bigcap \overline{T_n}$, none of its nodes need to be interrogated, when executing the message from node $P_n$ at a later instance. After describing the situations for case (b), it can be shown how node $P_t$ will behave towards the messages from $P_m$ and $P_n$ in case (c) in a rather similar way.

The number of extra interrogation messages and hence redundant parallel paths, which have been eliminated at node $P_t$ due to the later assumption, is equal to $FT(P_t) \bigcap \overline{T_m}$ or $FT(P_t) \bigcap \overline{T_n}$, for case (a), and to $\mid FT(P_t) \bigcap \overline{T_m} - FT(P_t) \bigcap \overline{T_n} \mid$, for cases (b) and (c), which in many cases, forms a considerable reduction in the number of diagnosis messages and diagnosis time as will be seen next.

An issue, which is linked to the consideration of preventing parallel paths is how a node

like $P_t$ should deal with the messages, whose execution will only cause repeating similar actions. This issue has been studied and, using the simulator, two different ways in which a node may act towards these messages have been investigated. These two ways are compared with each other and with the case in which a node is allowed to repeat similar actions without being restricted by the two aforementioned conditions. Hence, a total of three different ways were in fact considered and they have been designated as AC1, AC2 and AC3. The definitions of these three ways and results obtained by simulating the diagnosis of a number of fault situations using each one of them at a time will be introduced next.

AC1 : In this case, the node will execute every interrogation message on its queue independently without considering its previous actions. The factor, which will be considered by the node, however, is the contents of the set T. The criticism about this way is that a node is not restricted in the number of interrogation messages, that it should send to another node concerning a specific fault, hence redundant parallel paths may be generated.

AC2 : In this case, any interrogation message, whose execution will only result in repeating similar actions will be held up at the node, and whenever this node becomes qualified to send the reply back (i.e., either it receives a message of type PASS from at least one node or a message of type FAIL from all the nodes, that it had interrogated), it will issue, at a common time, replies to all interrogation messages, that are being held up by this node.

AC3 : This case is similar to AC2 in that the node does not repeat similar actions. However, it differs from the previous case in the sense that the messages, which are held up at the node, when AC2 is used are being answered back independently without delay(i.e., not at a common time, as in case AC2). By a way of contrast with the terms of the diagnosis algorithm, a node in this situation should be treated as if it has no fault-free testees to interrogate, and therefore it is required to send back a reply of type FAIL to its tester. This is unlike the previous case, where in this case, there will be a unique reply to every interrogation.

### 5.3.2 Results

The three different ways of AC1, AC2, and AC3 have been simulated in order to study our assumption concerning the prevention of redundant parallel paths and also to conclude on whether AC2 or AC3 should go with this assumption. A set of graph networks, which are constructed using the H-graphs (section 2.3) has been used, and at different positions in each network a single fault was simulated. For this simulated fault, the diagnosis process is applied using one of the three ways at a time. The graph networks, that have been used are $H(9,3;k_1,k_2,k_3)$ and $H(11,3;k_1,k_2,k_3)$. The single faults being simulated are designated as F1, F2, and F3. They correspond to the positions $k_1, k_2$, and $k_3$ of any node acting as a tester. Each time a fault is simulated, it is assumed to be the first in the system. The number of diagnosis messages and the diagnosis times will be considered for comparison. Any special case, such as failure to complete the diagnosis will be explained in further details. The number of diagnosis messages and diagnosis times and their corresponding plots for a set of $H(9,3;k_1,k_2,k_3)$ graph networks are shown in Tables(5.1 - 5.3) and Figures (5.11 - 5.13) respectively. Meanwhile, in Tables (5.4 - 5.6) and Figures (5.14 - 5.16), the diagnosis messages and diagnosis times and their corresponding plots for $H(11,3;k_1,k_2,k_3)$ graph networks are presented.

| | Diagnosis messages | | | Diagnosis time | | |
|---|---|---|---|---|---|---|
| $k_1, k_2, k_3$ | AC1 | AC2 | AC3 | AC1 | AC2 | AC3 |
| 1,2,3 | 145 | 117 | 123 | 52 | 45 | 38 |
| 1,2,4 | 109 | 102 | 104 | 43 | 41 | 33 |
| 1,2,6 | 109 | 105 | 107 | 39 | 39 | 37 |
| 1,2,7 | 109 | 102 | 104 | 44 | 40 | 35 |
| 1,2,8 | 60 | 60 | 60 | 25 | 25 | 25 |
| 1,3,4 | 101 | 101 | 101 | 27 | 27 | 27 |
| 1,3,6 | 68 | 68 | 68 | 28 | 28 | 28 |
| 1,3,7 | 60 | 60 | 60 | 19 | 19 | 19 |
| 1,3,8 | 105 | 105 | 105 | 32 | 32 | 32 |
| 1,4,6 | 60 | 60 | 60 | 19 | 19 | 19 |
| 1,4,7 | 162 | 120 | 126 | 56 | 44 | 29 |
| 1,6,7 | 94 | 94 | 94 | 27 | 27 | 27 |

Table 5.1 : Diagnosis messages and diagnosis times for $H(9,3;k_1,k_2,k_3)$ graph networks with simulated fault F1.

| | Diagnosis messages | | | Diagnosis time | | |
|---|---|---|---|---|---|---|
| $k_1, k_2, k_3$ | AC1 | AC2 | AC3 | AC1 | AC2 | AC3 |
| 1,2,3 | 102 | 102 | 102 | 22 | 22 | 22 |
| 1,2,4 | 94 | 94 | 94 | 19 | 19 | 19 |
| 1,2,6 | 98 | 98 | 98 | 19 | 19 | 19 |
| 1,2,7 | 98 | 91 | 93 | 23 | 23 | 23 |
| 1,2,8 | 80 | 80 | 80 | 22 | 22 | 22 |
| 1,3,4 | 102 | 98 | 100 | 32 | 32 | 32 |
| 1,3,6 | 88 | 88 | 88 | 22 | 22 | 22 |
| 1,3,7 | 110 | 102 | 104 | 38 | 35 | 28 |
| 1,3,8 | 94 | 94 | 94 | 30 | 30 | 30 |
| 1,4,6 | 94 | 94 | 94 | 29 | 29 | 29 |
| 1,4,7 | 162 | 120 | 126 | 56 | 44 | 29 |
| 1,6,7 | 95 | 91 | 93 | 29 | 28 | 28 |

Table 5.2 : Diagnosis messages and diagnosis times for $H(9,3;k_1,k_2,k_3)$ graph networks with simulated fault F2.

| | Diagnosis messages | | | Diagnosis time | | |
|---|---|---|---|---|---|---|
| $k_1, k_2, k_3$ | AC1 | AC2 | AC3 | AC1 | AC2 | AC3 |
| 1,2,3 | 60 | 60 | 60 | 22 | 22 | 22 |
| 1,2,4 | 84 | 84 | 84 | 22 | 22 | 22 |
| 1,2,6 | 123 | 109 | 113 | 44 | 38 | 36 |
| 1,2,7 | 116 | 102 | 106 | 47 | 41 | 34 |
| 1,2,8 | 131 | 99 | 105 | 54 | 45 | 34 |
| 1,3,4 | 60 | 60 | 60 | 19 | 19 | 19 |
| 1,3,6 | 98 | 98 | 98 | 19 | 19 | 19 |
| 1,3,7 | 101 | 101 | 101 | 29 | 29 | 29 |
| 1,3,8 | 130 | 109 | 113 | 48 | 40 | 38 |
| 1,4,6 | 104 | - | 99 | 37 | - | 32 |
| 1,4,7 | 162 | 120 | 126 | 56 | 44 | 29 |
| 1,6,7 | 60 | 60 | 60 | 19 | 19 | 19 |

Table 5.3 : Diagnosis messages and diagnosis times for $H(9,3;k_1,k_2,k_3)$ graph networks with simulated fault F3.

| | Diagnosis messages | | | Diagnosis time | | |
|---|---|---|---|---|---|---|
| $k_1, k_2, k_3$ | AC1 | AC2 | AC3 | AC1 | AC2 | AC3 |
| 1,2,3 | 264 | 153 | 167 | 78 | 51 | 36 |
| 1,2,4 | 178 | 141 | 151 | 51 | 48 | 37 |
| 1,2,5 | 151 | 133 | 137 | 49 | 41 | 33 |
| 1,2,7 | 187 | 144 | 152 | 57 | 44 | 38 |
| 1,2,8 | 143 | 141 | 135 | 44 | 50 | 32 |
| 1,2,9 | 139 | 135 | 137 | 45 | 44 | 42 |
| 1,2,10 | 72 | 72 | 72 | 28 | 28 | 28 |
| 1,3,4 | 172 | - | 138 | 49 | - | 34 |
| 1,3,5 | 197 | 144 | 152 | 63 | 47 | 31 |
| 1,3,8 | 141 | 126 | 128 | 45 | 41 | 41 |
| 1,3,10 | 134 | 126 | 130 | 35 | 35 | 35 |
| 1,5,7 | 72 | 72 | 72 | 22 | 22 | 22 |
| 1,5,8 | 205 | 145 | 155 | 66 | 50 | 46 |
| 1,8,9 | 163 | - | 142 | 54 | - | 38 |

Table 5.4 : Diagnosis messages and diagnosis times for $H(11,3;k_1,k_2,k_3)$ graph networks with simulated fault F1.

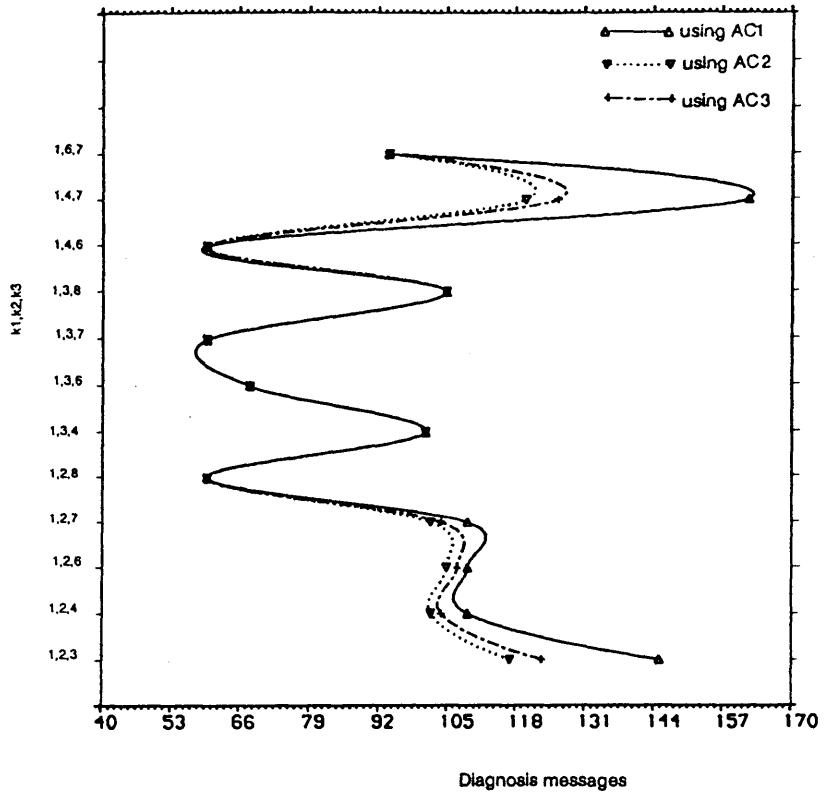
| | Diagnosis messages | | | Diagnosis time | | |
|---|---|---|---|---|---|---|
| $k_1, k_2, k_3$ | AC1 | AC2 | AC3 | AC1 | AC2 | AC3 |
| 1,2,3 | 163 | 183 | 144 | 26 | 26 | 26 |
| 1,2,4 | 134 | 126 | 128 | 22 | 22 | 22 |
| 1,2,5 | 126 | 115 | 117 | 22 | 22 | 22 |
| 1,2,7 | 162 | 134 | 142 | 23 | 23 | 23 |
| 1,2,8 | 116 | 112 | 114 | 23 | 22 | 22 |
| 1,2,9 | 133 | 115 | 119 | 26 | 26 | 26 |
| 1,2,10 | 100 | 100 | 100 | 25 | 25 | 25 |
| 1,3,4 | 176 | 133 | 137 | 57 | 45 | 36 |
| 1,3,5 | 162 | 140 | 144 | 46 | 40 | 40 |
| 1,3,8 | 171 | 141 | 145 | 51 | 43 | 36 |
| 1,3,10 | 114 | 114 | 114 | 30 | 30 | 30 |
| 1,5,7 | 151 | 126 | 130 | 46 | 40 | 34 |
| 1,5,8 | 162 | 134 | 140 | 23 | 23 | 23 |
| 1,8,9 | 127 | 119 | 121 | 41 | 37 | 35 |

Table 5.5 : Diagnosis messages and diagnosis times for $H(11,3;k_1,k_2,k_3)$ graph networks with simulated fault F2.

| | Diagnosis messages | | | Diagnosis time | | |
|---|---|---|---|---|---|---|
| $k_1, k_2, k_3$ | AC1 | AC2 | AC3 | AC1 | AC2 | AC3 |
| 1,2,3 | 72 | 72 | 72 | 25 | 25 | 25 |
| 1,2,4 | 112 | 112 | 112 | 22 | 22 | 22 |
| 1,2,5 | 163 | 130 | 136 | 59 | 48 | 37 |
| 1,2,7 | 205 | 145 | 157 | 66 | 50 | 43 |
| 1,2,8 | 176 | 137 | 145 | 59 | 48 | 39 |
| 1,2,9 | 184 | 130 | 138 | 68 | 53 | 43 |
| 1,2,10 | 214 | 127 | 137 | 81 | 57 | 43 |
| 1,3,4 | 72 | 72 | 72 | 22 | 22 | 22 |
| 1,3,5 | 181 | 133 | 137 | 65 | 46 | 37 |
| 1,3,8 | 158 | 137 | 141 | 46 | 40 | 38 |
| 1,3,10 | 189 | 130 | 138 | 66 | 49 | 37 |
| 1,5,7 | 174 | 119 | 121 | 58 | 38 | 37 |
| 1,5,8 | 191 | - | 158 | 56 | - | 38 |
| 1,8,9 | 72 | 72 | 72 | 22 | 22 | 22 |

Table 5.6 : Diagnosis messages and diagnosis times for $H(11,3;k_1,k_2,k_3)$ graph networks with simulated fault F3.

(a)




(b)

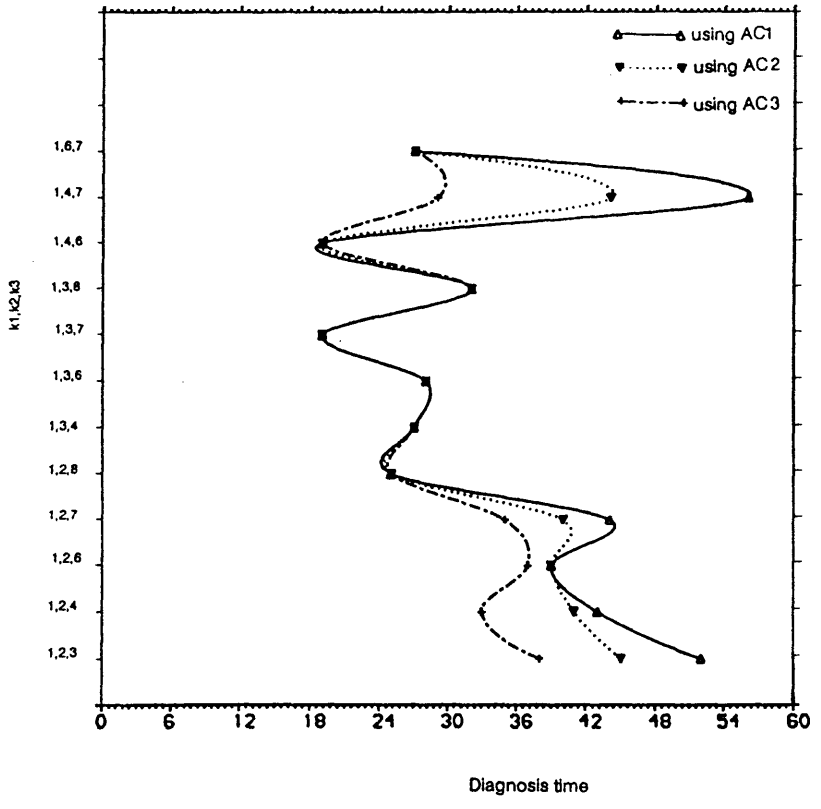
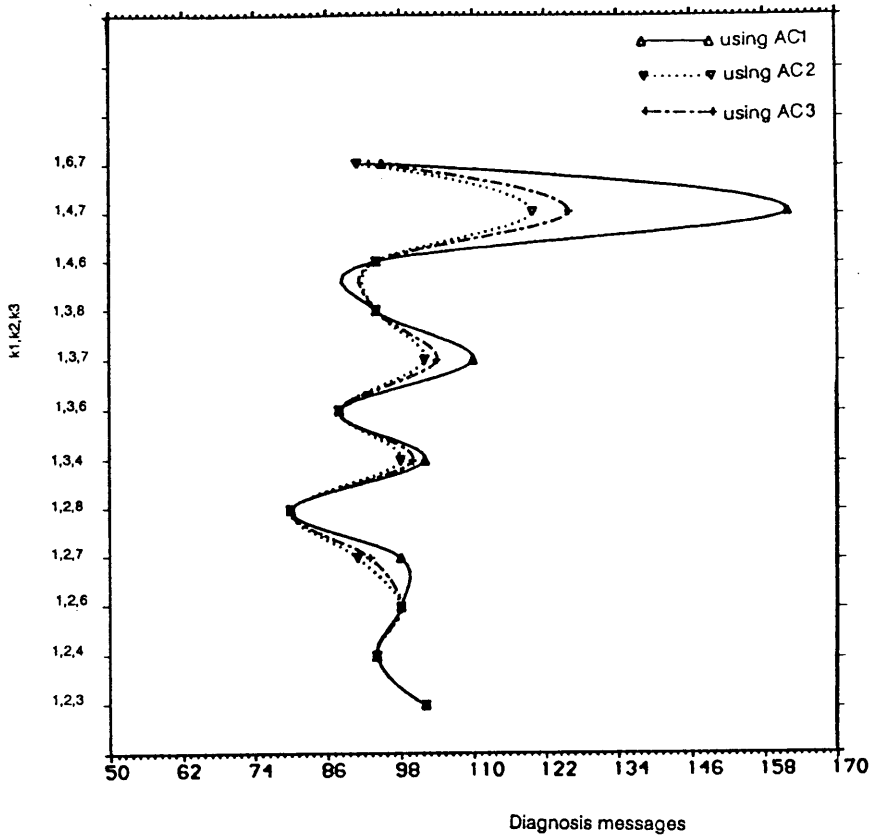



Figure 5.11: Plots for the data of Table 5.1.

(a)




(b)

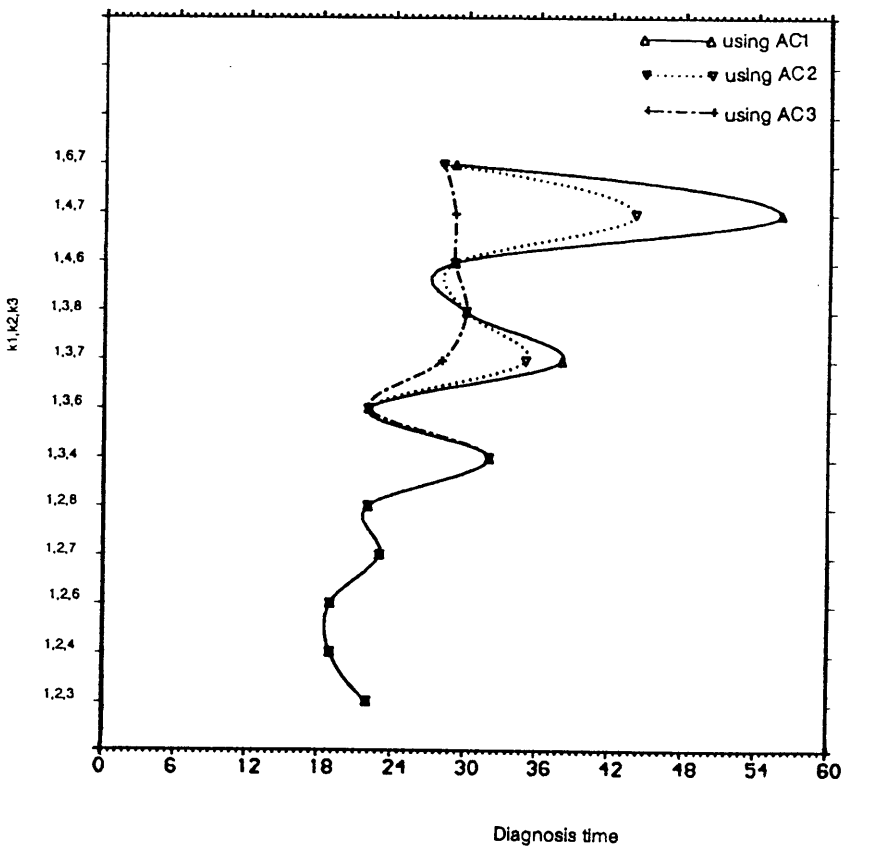
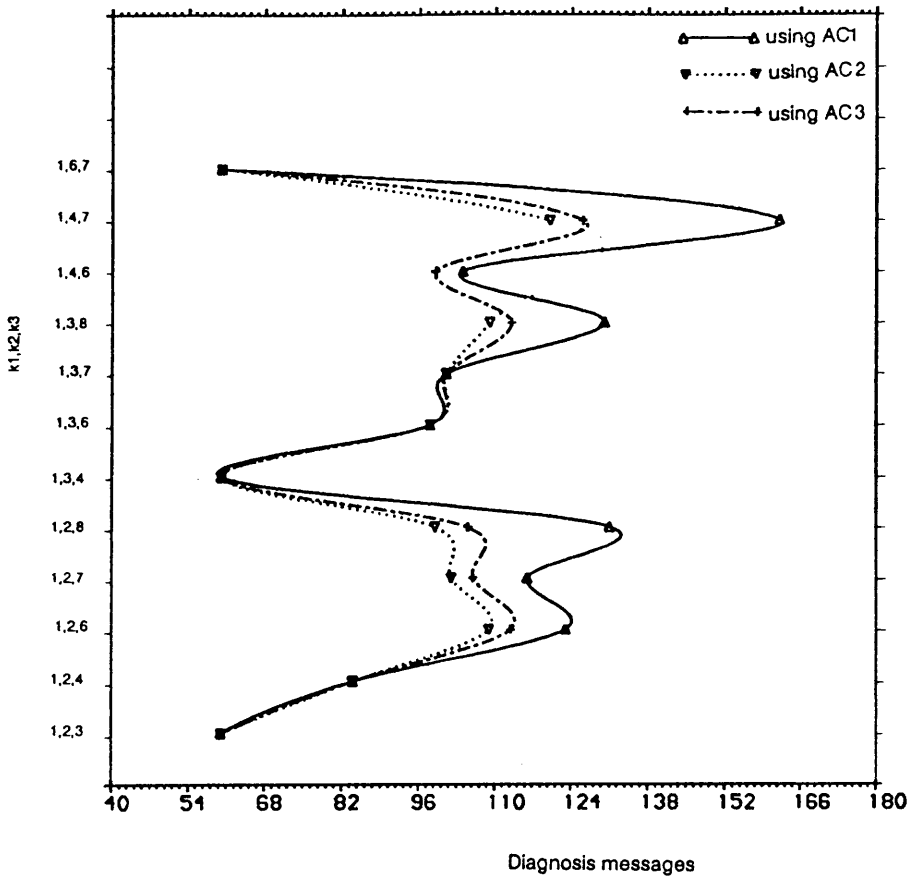



Figure 5.12: Plots for the data of Table 5.2.

(a)




(b)

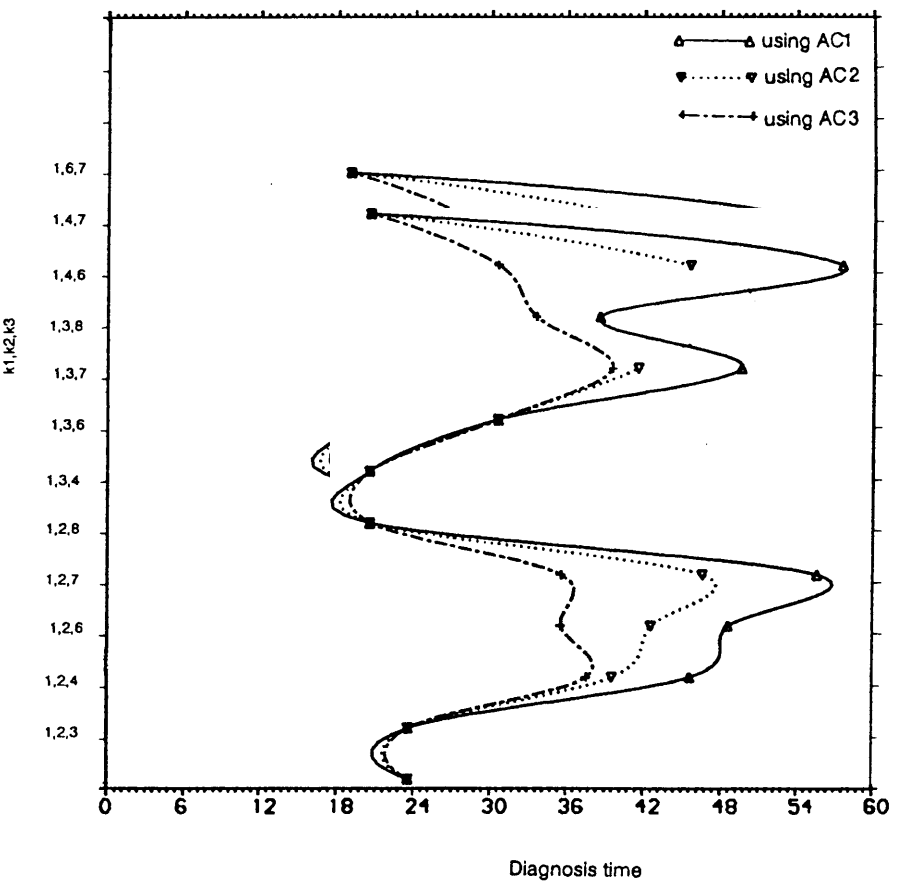



Figure 5.13: Plots for the data of Table 5.3.

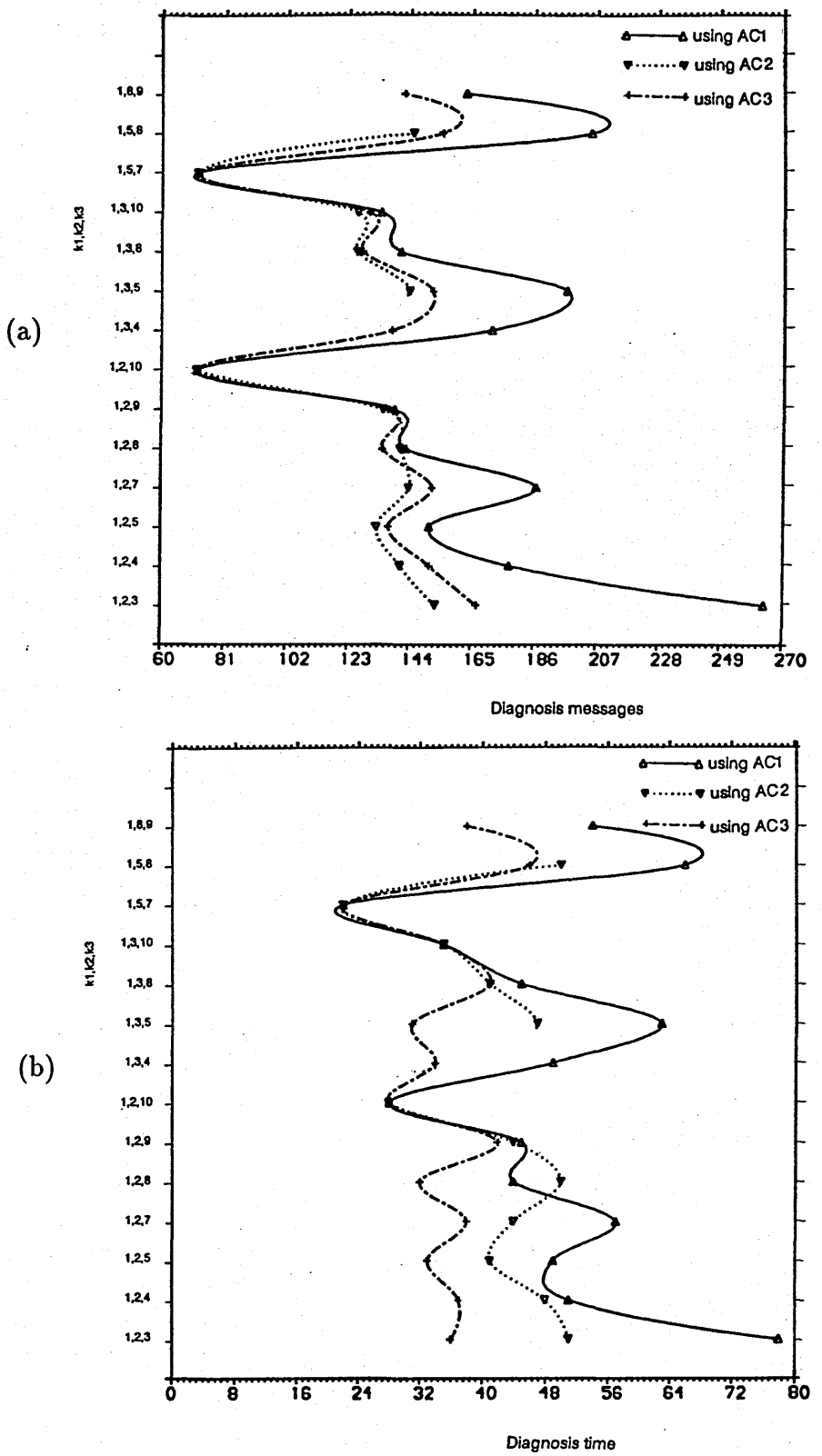
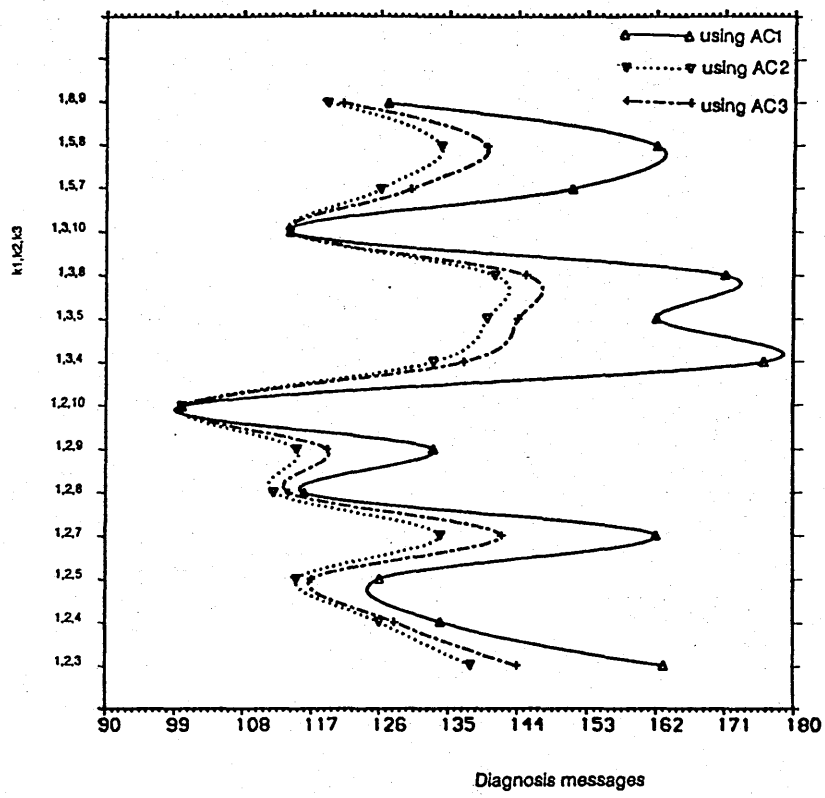



Figure 5.14: Plots for the data of Table 5.4.

(a)




(b)

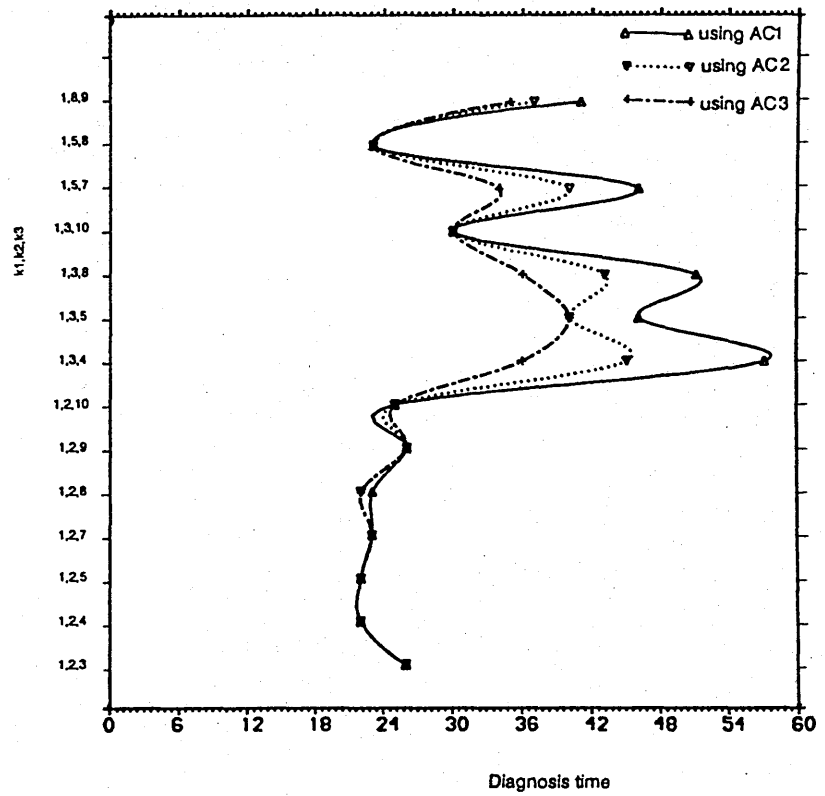
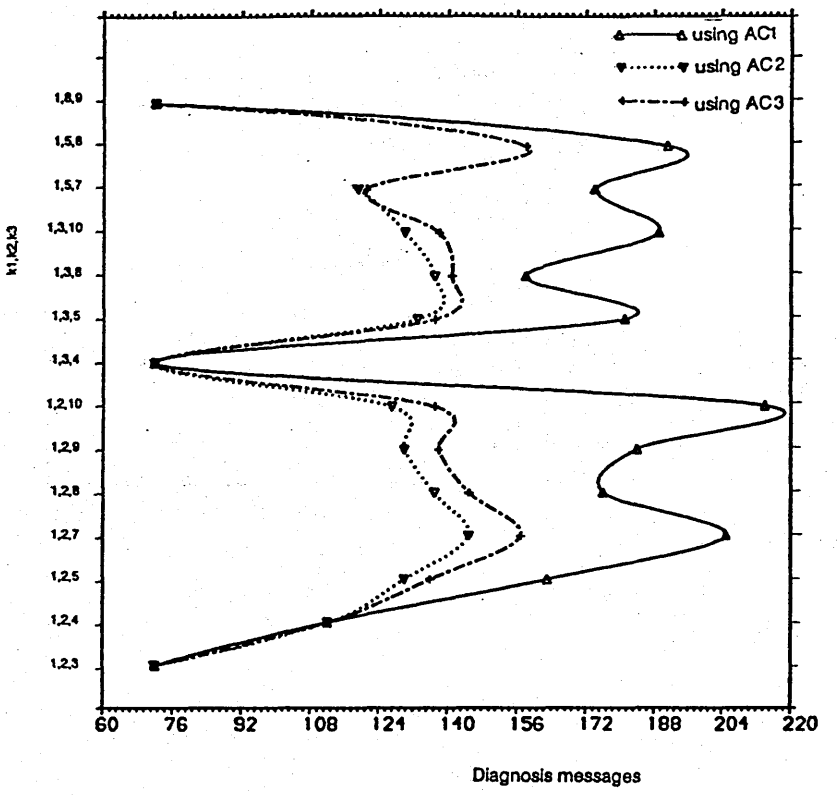



Figure 5.15: Plots for the data of Table 5.5.

(a)




(b)

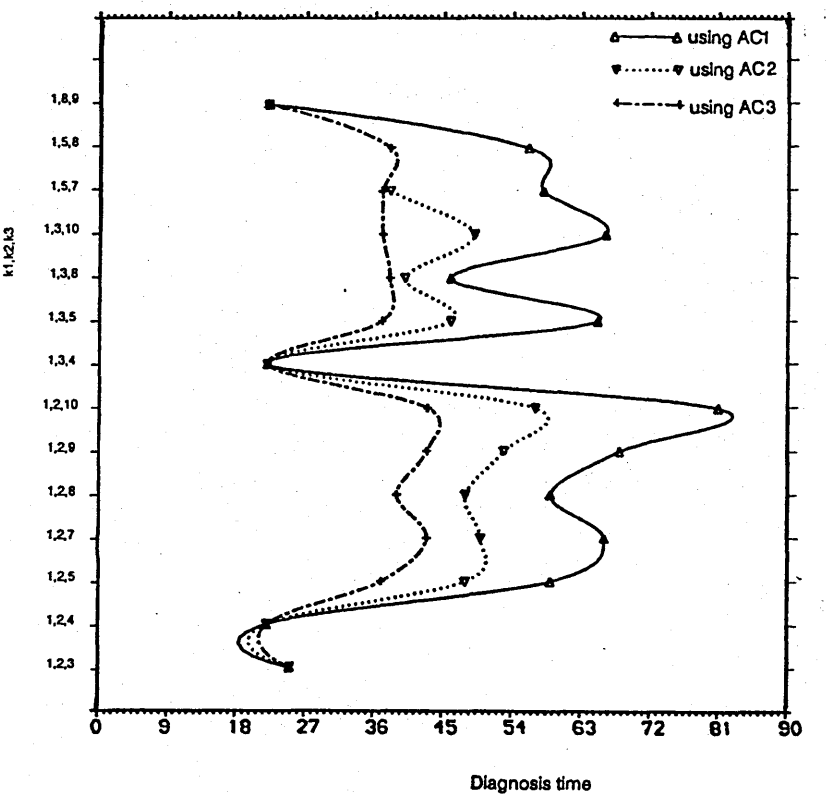



Figure 5.16: Plots for the data of Table 5.6.

In the previous results, there have been few cases, where the diagnosis of the simulated fault could not be achieved, while using AC2. These cases are:

(i) H(9,3;1,4,6) with simulated fault F3 (Table 5.3).

(ii) H(11,3;1,3,4) with simulated fault F1 (Table 5.4).

(iii) H(11,3;1,8,9) with simulated fault F1 (Table 5.4).

(iv) H(11,3;1,5,8) with simulated fault F3 (Table 5.6).

In order to find the reason behind the difficulty in achieving the diagnosis of the simulated faults in these cases, the diagnosis phases of cases (i) and (ii) will be explained. It will be noticed that the reason behind all of them is basically the same.

Example 5.6 : Consider case (i). Since the nodes in the H(9,3;1,4,6) graph network, which is shown in Figure 5.17 are possessing cyclic symmetry and the fault is assumed to be the first, therefore, any node can be considered as a tester. Assume, therefore, that node 1 has detected a fault in node 7 after applying a test on this node. The messages produced during the different phases of the diagnosis are shown in Figure 5.18.

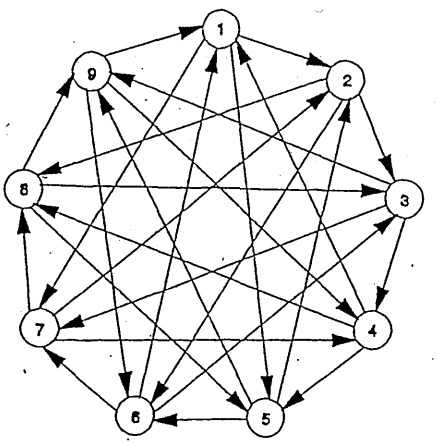



Figure 5.17: The H(9,3;1,4,6) graph network.

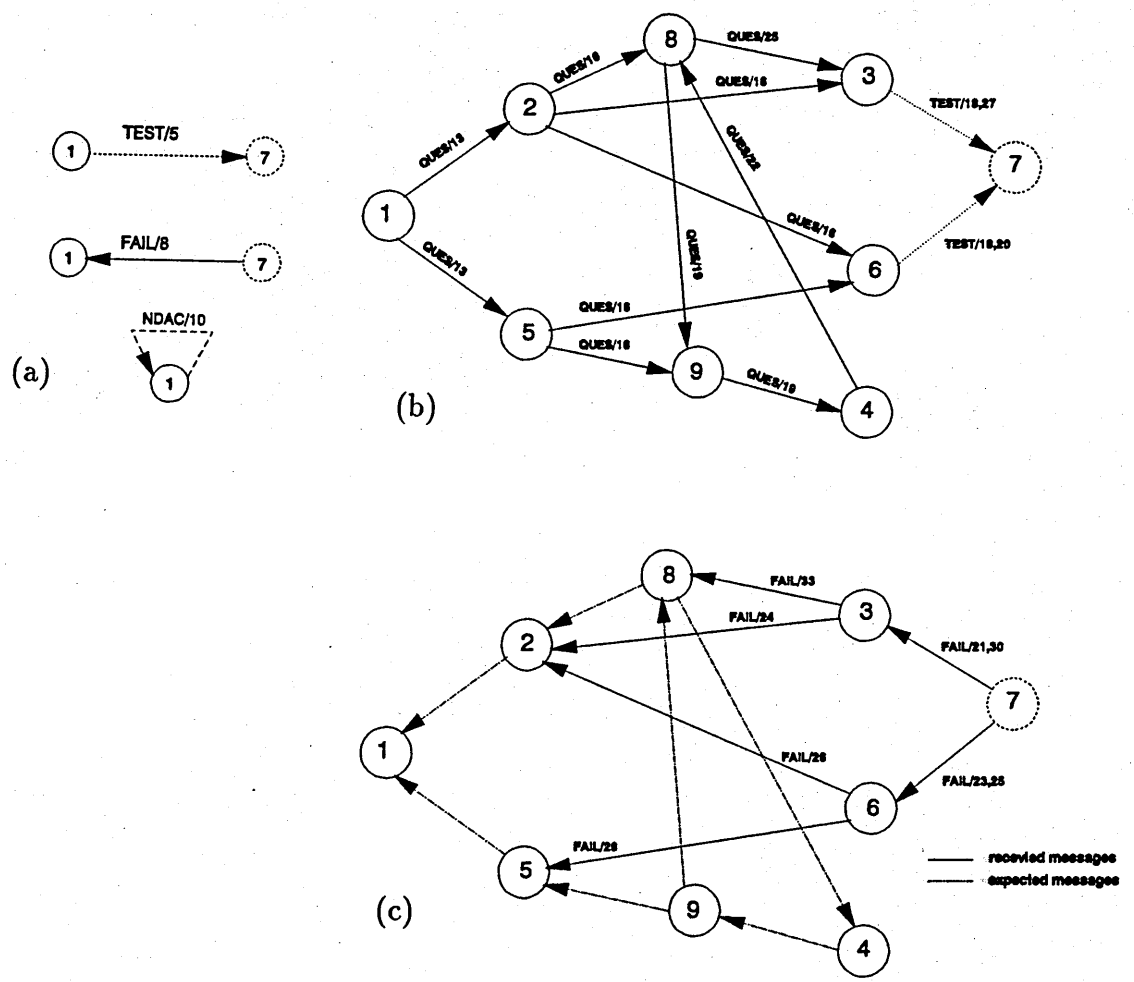



Figure 5.18: Uncompleted diagnosis phases of the H(9,3;1,4,6) graph network with simulated fault F3 (using AC2).

During the interrogation phase (Figure 5.18.b), the path *9-4-8-9* was generated, which is considered by the nodes involved as a cyclic interrogation path. The formation of this path has been due to the following sequence of events :

1. Node 8 received an interrogation message from node 2 with time stamp 16. This message carries a set T, which includes the following nodes T = {1,2,5,3,6,8}. In response to this message, node 8 has interrogated node 9.

2. An interrogation message was also received at node 9 from node 5 at the same simulated time (i.e., time stamp = 16). The set T of this message is T = {1,2,5,6,9}. Thus, node 4 was interrogated.

3. When node 9 started to execute the interrogation message from node 8, it has found, according to the set T of this message (T = {1,2,5,3,6,8,9}), that it should interrogate node 4. However, because it has already interrogated this node, and the two conditions assumed before are holding, it will not repeat the same action again. Instead, the message will be held up at the node so that no redundant parallel path will be generated and the message will be replied to at some later time.

4. From node 9, node 4 has received an interrogation message. The message contains a set T = {1,2,5,6,9,4} and its time stamp is 19. In response to this message, node 4 will interrogate node 8. The interrogation message, that will be received at node 8 will form the cyclic path *9-4-8-9.*

5. According to the contents of the set T of the message, which node 8 has received from node 4. Node 3 will be interrogated by node 8. Node 3 is a tester of the accused node (node 7).

As a reply to the interrogation message, that node 8 has sent to node 3, a message of type FAIL and time stamp 33 was received. This is shown in Figure 5.18.c. Because this message is of type FAIL and because node 8 has sent another interrogation message(to node 9 at 19), this node is not allowed to pass the message it has just received from node 3 unless a reply from node 9 is also received. Node 9 cannot reply before receiving a message from node 4, and node 4 cannot send this message unless itself receives a message from node 8. By this, the situation now comprises three nodes forming a cycle and each

cycle and each one is waiting for a message from the other before it can proceed with its diagnosis task. Thus, as far as the diagnosis process is concerned, the system is deadlocked. The *deadlock* in a computing system is defined as (the situation in which two or more programs are forever prevented from running to completion because their resource requirements are mutually exclusive [Cham80]).

**Example 5.7 :** Deadlock has also occurred in the other case (case(ii)). For this case, the H(11,3;1,3,4) graph network is shown in Figure 5.19, and it is assumed to have its first fault. The fault F1 is assumed to be simulated and node 1 is considered as the tester. Accordingly, node 2 has been accused. The application of the diagnosis algorithm and the phases it passes through are shown in Figure 5.20.

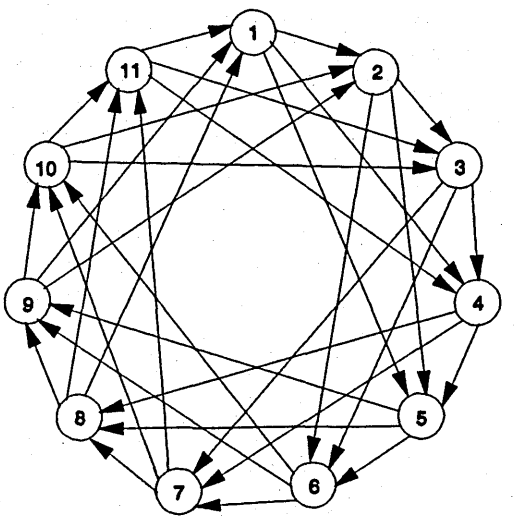



Figure 5.19: The H(11,3;1,3,4) graph network.

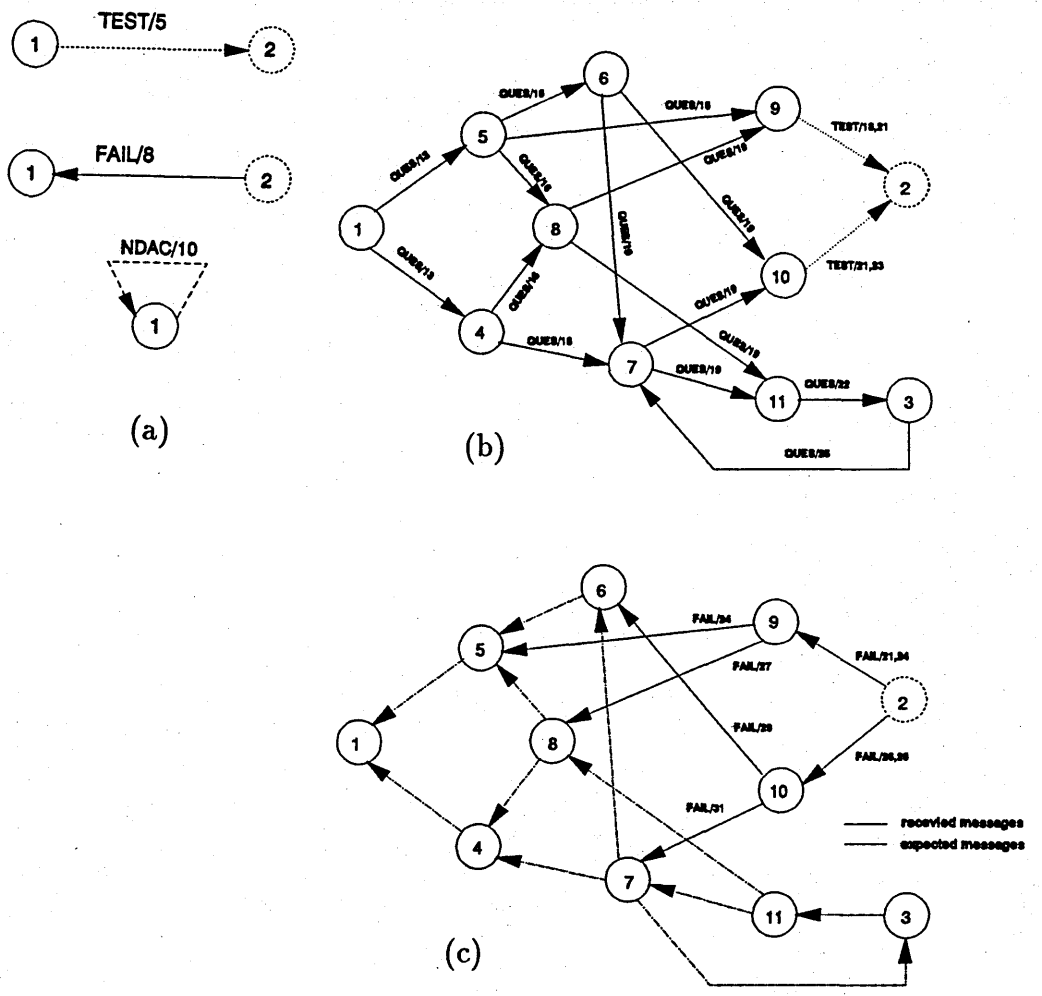



Figure 5.20: Uncompleted diagnosis phases of the H(11,3;1,3,4) graph network with simulated fault F1 (using AC2).

The path *7-11-3-7* is considered by the relevant nodes as cyclic and it is the reason behind the deadlock and eventually the failure to complete the diagnosis.

### 5.3.3 Comparison and Conclusions

By examining the plots in Figures (5.11-5.16), it is seen that the implementation of the algorithm in its original form (AC1) produces, in many cases, a number of messages which is considerably higher than that of the other two (AC2 and AC3) as well as requiring more time. It is when similar actions are repeated and hence redundant parallel paths are produced, that the differences appear. Both AC2 and AC3 were implemented in order to reduce the number of diagnosis messages and diagnosis times by restricting the nodes from repeating similar actions. The use of AC2 has resulted in a few cases, where diagnosis could not be performed. In AC2, replies to all interrogation messages, that are being held up at a node are assumed to be synchronized, once the node becomes ready. The implementation of this assumption has caused the formation of cyclic paths and eventually a deadlock.

AC3 has been used as an alternative to AC2, which is based on the fact that the edges of the cyclic paths causing the deadlock are basically related to a number of individual acyclic interrogation paths, and this fact should be considered by the nodes involved. Accordingly, a node that was used to hold up messages, whose execution will result in the formation of redundant parallel paths, when using AC2, must no longer hold up these messages. Instead, it should instantly return a message of type FAIL. Therefore, the interrogation message will be cleared from the node and the risk of it being part of a cyclic path in a later stage is removed.

A considerable reduction in the number of diagnosis messages has been achieved by using AC3 if compared to AC1, as well as a reduction in the diagnosis times with respect to both AC1 and AC2. It should be mentioned, however, that when the number of nodes in the system is relatively small or the interrogation phase contains no nodes, that are involved in more than one path, then all the cases AC1, AC2 and AC3 will converge to perform in a similar way. Among the three different ways, AC3 has been used in all the examples and descriptions included in this thesis, unless stated otherwise.

# Chapter 6

# Conclusions

## 6.1 Summary and Conclusions

Fault diagnosis is an integral part of fault-tolerance and forms an important tool in the maintenance strategy of distributed computer systems. The theory of fault diagnosis in distributed systems or system level fault diagnosis(as it usually called) has received considerable attention over the years. Several models have been proposed, in the literature, however, the fundamental model in the area was introduced by Preparata, Metze, and Chien [Prep67]. It was called the PMC model, and all subsequent work can be viewed in relationship to it.

Systems viewed according to the PMC and similar models are assumed to possess a global observer, which performs the diagnosis. In later work, however, the assumption of the global observer being capable of gathering test results and performing diagnosis without being itself subjected to faults was considered unrealistic [Kuhl80c]. The notion of distributed diagnosis was therefore introduced, in which the diagnosis tasks are assumed to be distributed among the system processors themselves and there is no global observer. In distributed diagnosis, each fault free processor can arrive at its own diagnosis by testing neighbouring nodes and receiving the results of tests that they have performed or obtained from other processors. Based on this approach, a number of diagnosis algorithms were proposed. In addition to diagnosing faulty processors in the system, some of these algorithms include a procedure for diagnosing link failures, and one of them , modified algorithm SELF3, has been considered, as a starting point in this thesis.

Although the PMC model and the distributed diagnosis model, share the idea of the testing graph and a set of identical basic assumptions about it (e.g., test invalidation, etc.), the diagnosability measures of the same graph are different in the two models. Quite generally a graph which is $t$-fault diagnosable on the PMC model is only $(t-1)$-fault diagnosable on the distributed diagnosis model. By viewing the distributed diagnosis

models in terms of the PMC model, an attempt has been made to reconcile this fact (section 3.8).

A software package has been written, in which a distributed system is simulated such that, each of its processors is mapped to a node in a simulated system. The simulated system is referred to as SIM and the software is designed to run on a single processor, which means that SIM can only carry out one action at a time. In order that a correct simulation is performed, a simulation algorithm was proposed. The main task of this algorithm is to allow each node in SIM to process messages in the same order as the equivalent processor in the real system. The simulator was designed to handle modifications of modified algorithm SELF3, and a number of assumptions and modifications were found necessary for its implementation in the software. One of these modifications is based on the fact that passing messages is the exclusive way of communication between nodes in SIM, therefore, any action that was not originally specified in the algorithm as a message, has been introduced to the simulator in a message format, and unlike the original algorithm all messages were modelled into a unified format, which was found convenient throughout the writing of the software, especially with the procedures containing message coding or decoding. The implementation of the proposed simulation algorithm requires the calculation of the simulated time. Because this is not considered in the diagnosis algorithm, each node is assumed to have a local clock, which represents this time. The value of the clock is appended to all messages produced by the node. It is being referred to as time stamp and used for the purpose of ordering.

Various types of network architectures are used for communication in distributed systems. Graph networks are among few types of communication architectures, that have been studied in this thesis. They represent a set of highly structured networks based on graphs. A class of graph networks, which are referred to as the H-graphs are examined. Their construction procedure and general properties are presented. These networks, which are regular and homogeneous, possess the property of cyclic symmetry. This particular property makes them look the same when viewed from any node and help in reducing the complexity of designing routing algorithms. These networks have been used to construct examples which explain the operation of the diagnosis algorithm in detection and location of faults in the system. The procedure of performing diagnosis has been classified into a number of phases as an attempt to make it more understandable.

Graphical representation of the paths followed by messages are also presented in which messages are defined by their types and time stamps. Similar examples for a selected testing assignment of the four-dimensional hypercube are given as well.

A study has been carried out to reduce the number of diagnosis messages and the time required to perform a diagnosis, especially in large systems, while preserving the correctness of the diagnosis. The study is based on eliminating any message whose role in the diagnosis will only be a repetition of another message. A considerable reduction in both the number of diagnosis messages and the values of diagnosis times are obtained, in many cases, compared to the values obtained when the steps recommended by the modified algorithm SELF3, as published, are followed. It is shown that some methods of removing redundant messages may lead to deadlock.

The following specific conclusions may be made from the simulated results:

- A system, which is theoretically $t$-fault diagnosable may in fact diagnose more than $t$-faults if these are located such that the set of fault-free nodes, which remain, form a subgraph of sufficient connectivity to perform the diagnosis.

- The accuser node is always assumed to be fault-free. If, however, this node is faulty and incorrectly accuses one of its fault-free testees of being faulty then modified algorithm SELF3 will not recognize this. This leads to the general conclusion that this algorithm does not cope with test invalidation.

- For a system comprising a set of equivalent processors, the time required for diagnosis generally depends upon the following:

  - The dimension of the system,

  - the size and distribution of the computational load being processed concurrently with the diagnosis, and

  - the lengths of the paths between the accuser and fault-free testers of the accused node.

- The graph networks $H(n, r; k_1, ..., k_r)$, described in section 2.3.2, provided they are connected, are generally $r$-fault diagnosable on the PMC model and $(r-1)_{n,l}$-fault self diagnosable using modified algorithm SELF3.

- The testing process is assumed to require the same time as any other diagnostic operation during the simulation. This assumption may not necessarily be correct in a practical system, where a comprehensive test requires a relatively long time. Because of this, we can conclude that the testing may initially restricted to a subset of links in the testing graph of the fault-free system, for the purpose of fault detection. In the $H(n, r; 1, k_2, ..., k_r)$ graph networks, the initial testing may be restricted to the single loop (i.e., the links $i \rightarrow i + 1$).

- The number of nodes that are involved in the diagnosis process of a given single fault is restricted to a subset of nodes iff every path in the testing graph from the accuser reaches a tester of the accused node. (e.g., testing over links $i \rightarrow i+2$ in the H(n,2;1,2) graph networks satisfies the condition, or links $i \rightarrow i+3$ in H(n,3;1,2,3).).

## 6.2 Suggestions for Further Work

1. An individual processor in the system is assumed to possess no knowledge about the topology of the system apart from its position with respect to its neighbours, according to modified algorithm SELF3. This assumption has led to the inclusion of the *set* $T$ to every interrogation message as a provision for the occurrence of cyclic paths. The use of set T did not result in a noticeable problem in the examples presented in this thesis because we are dealing with systems, which are relatively small. In large systems, however, this set may grow to contain a large number of nodes, and therefore its encoding as part of a message will not necessarily be convenient or even possible. An alternative, which might be investigated, is to assume that the processors of the system are provided with more information about its topology, and to design these information such that the use of set T is eliminated and a better routing of messages is implemented.

2. In modified algorithm SELF3, the accuser is the node that is given the responsibility of co-ordinating the diagnosis process without necessarily being tested itself. If in fact the accuser is faulty and has made an invalid test on the accused node, then the system will be involved in diagnosing the condition of a node depending on a false accusation. Because any accusation issued will cause a degradation in the system after performing the corresponding diagnosis, it is therefore important for

the diagnosis to result in isolating those elements, which are really faulty. One possibility which might help in this and therefore reduce the effect of invalid tests is to adapt the algorithm so that the accuser node is tested and then allowed to proceed with the diagnosis, only if it is found fault-free. Otherwise, its status as an accuser should be changed so that it will be the accused node, and its accusation must be confirmed by testing the node that has found it faulty (i.e., the new accuser) and so on. This suggestion is expected to lead to an algorithm having the same diagnosability measure as the PMC model.

3. The topological properties of the proposed H-graphs can be further investigated and compared with other graph networks, in a way that might be similar to Beivide et al [Beiv91], where a class of graph networks and a systematic procedure for constructing them are presented. It is interesting to note that, the graph networks proposed in this reference, can in fact be constructed in an easier way using Formula 2.1, in this thesis, as undirected $H(n, 2; b-1, b)$ graph networks, where $n$ is the number of nodes and $b = \lceil \sqrt{n/2} \rceil$.

4. In line with the objectives of the work presented in this thesis, which serve in reducing the gap between the theoretical results in the area of system level fault diagnosis and its applications in distributed systems, an experimental test bed can be built and used to implement the proposals made, and the results obtained. This step will certainly give new insights to the work and its results. A number of modifications and assumptions may be found necessary, and they may also need to be more specific, especially those regarding the execution of tests and the analysis of test results.

5. During the reply phase of the diagnosis process in modified algorithm SELF3 (chapter 5), there are two types of messages (PASS and FAIL) used to convey test results. The message of type PASS means that the node being tested (the accused node) is fault-free. The message of type FAIL, on the other hand, could either mean that the accused node is faulty or that a tester has not been found. These two possibilities are interpreted by the accuser as a confirmation for its accusation. The failure of the system to find a tester for the accused node may be caused by the system reaching or exceeding its diagnosability limit. We believe that it is important for

the system to be capable of recognizing non-diagnosable situations and hence when it exceeds its diagnosability. An additional message (say, NO-TEST) can be added to the set of messages used during the reply phase to help the system in recognizing non-diagnosability. At this stage, it is desirable to increase the diagnosability of the system and therefore make it more capable of diagnosing the fault that it has just failed to diagnose.

6. We suggest two ways of increasing diagnosability, both requiring further investigation. Firstly, by adding extra links to the testing graph the diagnosability of the system can be increased (i.e., using a dynamic testing graph rather than a fixed one). Secondly, a testing graph of hierarchical structure can be implemented, and when a subset of nodes fail to diagnose a fault, the diagnosis task moves to a higher level in the hierarchy.

7. The modified algorithm SELF3 specifies a sequence of messages which occur following the accusation of a single faulty node. In the event of multiple faults the diagnosis algorithm produces an independent sequence of messages relating to each accusation made. In general, further accusation may be made on nodes which have already been accused. Such accusations are redundant and their removal would be the subject for further investigation.

# References


**[Adam87]** - Adams III, G.B., Agrawal, D.P., and Seigl, H.J., *A Survey and Comparison of Fault Tolerant Multistage Interconnection Networks*, Computer, June 1987, pp.14-27.

**[Bars76]** - Barsi, F., Grandoni, F., and Maestrini, P., *A Theory of Diagnosability of Digital Systems* , IEEE Trans. on Computers, June 1976, pp. 585-593.

**[Batc80]** - Batcher, K.E., *Design of a Massively Parallel Processor* , IEEE Trans. on Computers, vol. 29, Sept. 1980, pp. 836-840.

**[Beiv91]** - Beivide, R., Herrada, E., Balcazar, J.L., and Arruabarrena, A., *Optimal Degree Networks of Low Degree for Parallel Computer* , IEEE Trans. on Computers, October 1991, pp. 1109-1124.

**[Bhuy84]** - Bhuyn, L., and Agrawal, D.P., *Generalized Hypercube and Hyperbus Structure for a Computer Network*, IEEE Trans. on Computers, April 1984, pp.323-333.

**[Bian90]** - Bianchini, R., Goodwin, K., and Nydick, D.S., *Practical Application and Implementation of Distributed System-Level Diagnosis Theory* , in Proc. of The 20th Symposium on Fault Tolerant Computing, 1990, pp. 332-338.

**[Cham80]** - Champine, G.A., Coop, R.D., Heinselman, R.C., *Distributed Computing Systems, Impact on Mangment, Design and Analysis*, North-Holland Publishing Company 1980.

**[Chen90]** - Chen, M.S., Shin,K.G., and Kandlur, D.D., *Addressing, Routing, and Broadcasting in Hexagonal Mesh Multiprocessors*, IEEE Trans. on Computers, January 1990, pp.10- 18.

**[Dahb84]** - Dahbura, A., and Masson, G.M., *An $O(n^{2.5})$ Fault Identification Algorithm for Diagnosable Systems*, IEEE Trans. on Computers, June 1984, pp.486-492.

**[Dahb86]** - Dahbura, A.T., *An Efficient Algorithm for Identifying The Most Likely Fault Set in a Probabilistically Diagnosable System* , IEEE Trans. on Computers, April 1986, pp. 354-356.

**[Dahb87]** - Dahbura, A., *System Level Fault Diagnosis: A Perspective for The Third Decade*, In Proceedings of the 1987 Princeton Workshop on Algorithm, Architecture, and Technology, Princeton University, NJ, pp.411-434.

**[Davi79]** - Davies, D.W., Barber, D.L.A, Price, W.L., and Solomonides, C.M., *Computer Networks and Their Protocols*, John Wiley and Sons, UK, 1979.

**[Digi78]** - *Digital Time Division Command/Response Multiplex Data Bus*, MIL STD-1553B, Sept. 1978.

**[Dolt91]** - Dolter, J.W., Ramanathan, P., and Shin, K.G., *Performance Analysis of Virtual Cut-Through Switching in HARTS: A Hexagonal Mesh Multicomputer*, IEEE Trans. on Computers, June 1991, pp.669-680.

**[Farb72]** - Farber, D.J., Larson, K.C., *The System Architecture of The Distributed Computer System-The Communication System*, Symp. on Computer Communication Networks and Telegraphic, Brooklyn Polytechnic Press, April 1972, pp.21-27.

**[Frie75]** - Friedman, A.D., *A New Measure of Digital System Diagnosis*, The 6th International Symposium on Fault Tolerant Computing, 1975, pp.167-169.

**[Frie80]** - Friedman, A.D., and Simoncini, L., *System Level Fault Diagnosis*, Computer, March 1980, pp.47-53.

**[Ghaf89]** - Ghafoor, A. and Bashkow, T.R., *Bisectional Fault Tolerant Computation Architecture for Supercomputer Systems*, IEEE Trans. on Computers, October 1989, pp. 1425-1446.

**[Gibb85]** - Gibbons, A., *Algorithmic Graph Theory*, Cambridge University Press, UK,1985.

**[Grif86]** - Griffith, G.W., *A Fault Tolerant Distributed Computer System for Automotive Applications*, M.Sc. Thesis, Cranfield Institute of Technology, UK, 1986.

**[Haki81]** - Hakimi, S.L., and Chwa, K.Y., *Schemes for Fault Tolerant Computing : A Comparison of Modularly Redundant and t-diagnosable Systems*, Information and Control, vol.-49, June 1981, pp.212-238.

**[Haki84]** - Hakimi, S.L., and Nakajima, K., *On Adaptive System Diagnosis*, IEEE Trans. on Computers, March 1984, pp.234-240.

**[Holt81]** - Holt, C.S., and Smith, J.E., *Diagnosis of Systems with Asymmetric Invalidation* , IEEE Trans. on Computers, Sept. 1981, pp. 679-690.

**[Hopk78]** - Hopkins, Jr, A.L., Smith, III, T.B., and Lala, J.H., *FTMP-A Highly Reliable Fault Tolerant Multiprocessor for Aircraft*, Proceedings of the IEEE, October 1978, pp. 1221-1239.

**[Hoss84]** - Hosseini, S.H., Kuhl, J.G., Reddy S.M., *A Distributed Algorithm for Distributed Computing Systems with Dynamic Failure and Repair* , IEEE Trans. on comput., March 1984, pp.223-233.

**[Hoss88]** - Hosseini, S.H., Kuhl, J.G., Reddy S.M., *On Self- Fault Diagnosis on Distributed Systems* , IEEE Trans. on Computers., February 1988, pp. 248-251.

**[IEEE85]** - *IEEE 802.4, Token Passing Bus Access Method*, IEEE Press, New York, 1985.

**[Karu79]** - Karunanithi, S., and Friedman, A.D., *Analysis of Digital Systems Using a New Measure of System Diagnosis*, IEEE Trans. on Computers, Feb. 1979, pp.121-133.

**[Kim79]** - Kim, K., *Error Detection, Reconfiguration and Testing in Distributed Computing Systems.* , in Proc. First Int. Conf. on Distributed Systems, October 1979, pp.284-295.

**[Kreu87]** - Kreutzer, S.E., and Hakimi, S.L., *System Level Fault Diagnosis: A Survey.*, Microprocessing and Microprogramming, May 1987, pp.323-330.

**[Kuhl80a]** - Kuhl, J.G., and Reddy, S.M., *Distributed Fault Tolerance for Large Multiprocessor Systems*, IEEE Symposium on Computer Architecture, 1980, pp.23-30.

**[Kuhl80b]** - Kuhl, J.G., and Reddy, S.M., *Some Extensions to The Theory of System Level Fault Diagnosis*, 10th Int. Confer. on Fault Tolerant Computing, October 1980, pp.291-296.

**[Kuhl80c]** - Kuhl, J.G., *Fault Diagnosis in Computing Networks* , PhD Thesis, University of Iowa, 1980.

**[Kuhl81]** - Kuhl, J.G., Reddy, S.M., *Fault Diagnosis in Fully Distributed Systems* , IEEE Symposium on Fault Tolerant Computing, 1981, pp. 100-105.

**[Kuma87]** - Kumar, V.P., and Reddy, S.M., *Augmented Shuffle-Exchange Multistage Interconnection Networks*, Computer, June 1987, pp.30-40.

**[Maen81]** - Maeng, J., and Malek, M., *A Comparison Connections of Multiprocessor Systems.*, Proc. of Int. Conference on Fault Tolerant Computing, June 1981, pp.173-175.

**[Mahe76]** - Maheshwari, S.N., and Hakimi, S.L., *On Modules for Diagnosable Systems and Probabilistic Fault Diagnosis*, IEEE Trans. on Computers, March 1976, pp.228-236.

**[Mall78]** - Mallela, S.M., and Masson, G.M., *Diagnosable Systems for Intermittent Faults*, IEEE Trans. on Computers, June 1978, pp.560-566.

**[Meye78]** - Meyer, G.G.L., and Masson, G.M., *An Efficient Fault Diagnosis Algorithm for Symmetric Multiple Processor Architectures*, IEEE Trans. on Computers, November 1978, pp.1059-1063.

**[Mudg87]** - Mudge, T.N., Hayes, T.P., and Winsor, D.C., *Multiple Bus Architectures*, Computer, June 1987, pp.42-48.

**[Nair78]** - Nair, R.K., *Diagnosis, Self Diagnosis, and Roving Diagnosis in Distributed Digital Systems*, PhD Thesis, University of Illions, 1978.

**[Naka81]** - Nakajima, K., *A New Approach to System Diagnosis*, In Proc. 19th Annual Allerton Confer. Commun. Control and Computers, Sept. 1981, pp.697-706.

**[Nova87]** - Novak, F., and Gyergyek, L., *Distributed System Diagnosability based on Self Testing System Nodes* , Microprocessing and Microprogramming, August 1987, pp.489-496.

**[Pete85]** - Peters, J.F., *Problem Solving with PASCAL, Programming Methods, Algorithms, and Data Structures*, CBS College Publishing, Japan 1986.

**[Prad82]** - Pradhan, D.K., and Reddy, S.M., *A Fault Tolerant Communication Architecture for Distributed Systems*, IEEE Trans. on Computers, September 1982, pp.863-870.

**[Prad85]** - Pradhan, D.K., *Fault-Tolerant Multiprocessor Link and Bus Architectures*, IEEE Trans. on Computers, January 1985, pp.33-45.

**[Prad86]** - Pradhan, D.K., *Fault-Tolerant Computing: Theory and Techniques*, Englewood Cliffs, NJ, 1986.

**[Prep67]** - Preparata, F.P., Metze, G., and Chien R.T., *On the Connection Assignment Problem of Diagnosable Systems* , IEEE Trans. on Electro. Comput., December 1967, pp.848-854.

**[Rand78]** - Randell, B., and Treleavan, P., *Reliability Issues in Computing System Design*, Computing Surveys, vol.10, June 1978, pp.123-165.

**[Rang88]** - Rangarajan, S., and Fussell, D., *A Probabilistic method for Fault Diagnosis of Multiprocessor Systems*, 18th Int. Symposium on Fault Tolerant Computing, June 1988, Tokyo, pp.278-283.

**[Righ98]** - Righter, R., and Walrand, J.C., *Distributed Simulation of Discrete Event Systems*, Proceedings of The IEEE, January 1989, pp.99-113.

**[Russ75a]** - Russell, J.D., and Kime, C.R., *System Fault Diagnosis : Closure and Diagnosability without Repair.*, IEEE Trans. on Computers, November 1975, pp.1078-1089.

**[Russ75b]** - Russell, J.D., and Kime, C.R., *System Fault Diagnosis : Masking, Exposure, and Diagnosability without Repair.*, IEEE Trans. on Computers, December 1975, pp.1155-1161.

**[Sawc87]** - Sawchuk, A.A., Jenkins, B.K., Raghavendra, C.S., and Varma, A., *Optical Crossbar Networks*, Computer, June 1987, pp.50-60.

**[Sieg79]** - Siegal, H.J., McMillen, R.J., and Mueller, Jr, P.T, *A Survey of Interconnection Methods for Reconfigurable Parallel Processing Systems*, AFIPS Conf. Proceedings, vol.48, 1979, NCC, pp.387-400.

**[Sieg85]** - Siegel, H.J., *Interconnection Networks for Large Scale Parallel Processing: Theory and Case Studies*, D.C Heath and Company, USA, 1985.

**[Skil84]** - Skilton, F.R., *Understanding PASCAL*, Wm. C. Brwon Publishers, USA, 1984.

**[Slom87]** - Sloman, M., and Kramer, J., *Distributed Systems and Computer Networks*, Printice-Hall International, UK, 1987.

**[Sull77]** - Sullivan, H., Bashkow, T., and Klappholz, D., *A Large Scale Homogenous, Fully Distributed Parallel Machine* , Proc. Fourth Annual Symposium on Computer Architecture, March 1977, pp.105-124.

**[Sull86]** - Sullivan, G.F., *The Complexity of System Level Diagnosis and Diagnosability*, PhD Thesis, Yale University, December 1986.

**[Swan77]** - Swan, R.J., Fuller, S.H., and Siewioreck, D.P., *Cm* - A modular, multiprocessor*, 1977 AFIPS Conference Proceedings, vol. 46, pp.637-644.

**[Thur74]** - Thurber, K.J., *Interconnection networks - A Survey and Assessment*, AFIPS Conf. proceedings, vol. 43, 1974, NCCP, pp.909-919.

**[Wens78]** - Wensely, J.H. et al, *SIFT: Design and Analysis of Fault Tolerant Computer for Aircraft Control*, Proceedings of the IEEE, vol. 66, October 1978, pp.1240-1255.

**[Witt78]** - Wittie, L.W., *MICRONET : A Reconfigurable Microcomputer Network For Distributed System Research* , Simulation, vol. 31, pp. 145-153, November 1978.

**[Vrie90]** - De Vries, R.C., *Reducing Null Messages in Misra's Distributed Discrete Event Simulation Method*, IEEE Trans. on Software Engineering, vol. 16, No. 1, January 1990, pp. 82-91.

**[Yang86]** - Yang, C.L., Masson, G.M., and Leonetti, R.A., *On Fault Isolation and Identification in t/t Diagnosable Systems*, IEEE Trans. on Computers, July 1986, pp. 639-643.

# Appendix A

# Survey of The Procedures in The Software

In chapter (4), it was mentioned that the software is composed of four modules. These are Input, Initialization, Simulation, and Output modules. The main function of each one of these modules was described in section 4.3. Each module is responsible of activating its sub-modules. The sub-modules are usually composed of a number of procedures, and each procedure is a unit, that carries out a specific task. The naming of these procedures as well as the variables are intended to be task descriptive and to make them self-apparent.

The main modules are activated and called by the main program to be executed in the sequence that was shown in Figure 4.2. In this Appendix, a structure chart of the main program and its four modules are presented. Most of the procedures in the software, which are identified by their programming names are summarized in terms of their operation tasks. Moreover, some selected procedures are described by their operational sequence charts.

## A.1 The Main Program

The function of the main program is to call the four modules for execution in sequence and then terminate the execution. Its structure chart is shown in Figure A.1.

## A.2 The Input Module

The structure chart of this module is shown in Figure A.2. It calls the following procedures;

### A.2.1 : ch-gr-file and ch-fl-file

Each one of these procedures gives the user a choice either to let the input data (network and fault specification) to be read from a file, that is already prepared in the directory or for a new file to be created. If the first option is chosen then procedure 'ex-inp-file' will be called, while procedure 'cr-inp-file' is called if the second option is chosen.

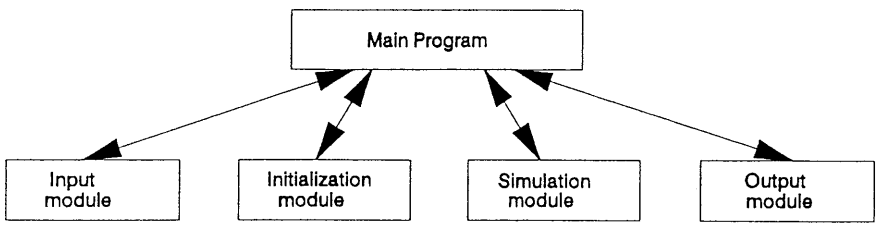



Figure A.1 : Structure chart of the main program

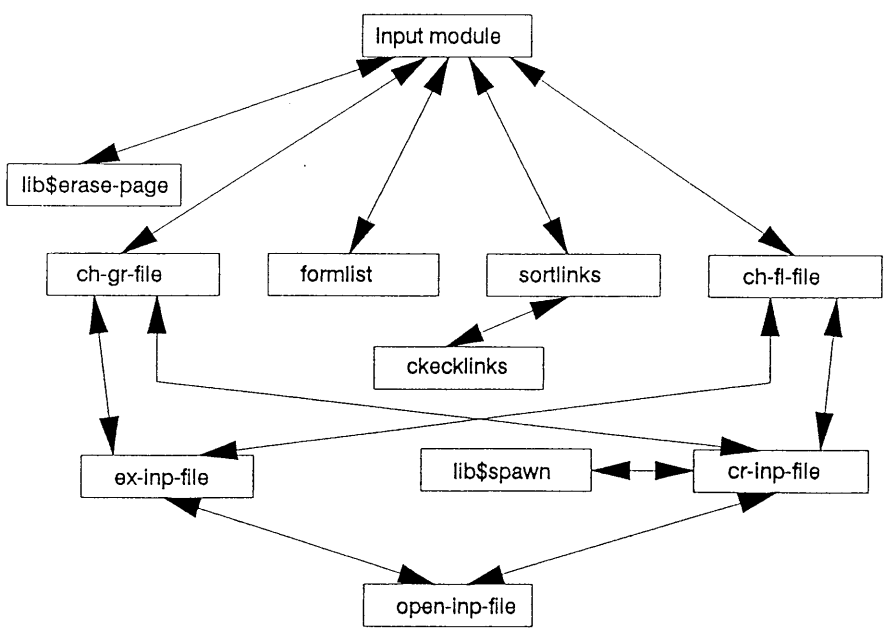



Figure A.2 : Structure chart of the Input module

### A.2.2 : lib$spawn

This procedure is called by 'cr-inp-file' and by some other procedures. It is a utility routine in the VAX computer and it can be used for opening a subprocess and sending a command text to the VMS operating system, during the execution of the program, for editing a new file.

### A.2.3 : formlist

The function of this procedure is to form a linked list of records for the system nodes as well as a record for the bus, where the system is assumed to use a shared bus for communication. Each record is composed of a number of fields and most of these fields are initialized in this procedure even though some of them may be re-initialized later in the program.

### A.2.4 : sortlinks, checklinks

In procedure 'sortlinks', fields of each node record are provided with the information, that is assumed to be known by the node about the architecture of the system in terms of testing assignments. For every node $x$, this information is collected from the network specification and sorted out in two separate fields. One of these fields is to contain the addresses of the nodes, that will test node $x$ and the links through which these tests are conducted. The second field is to contain the addresses of the nodes that will be tested by node $x$ and the links through which the tests are conducted. Procedure 'sortlinks' will call procedure 'checklinks'.

### A.2.5 : lib$erase-page

This procedure, which is a utility routine in the VAX computer, is called by many other procedures in the package. Its function is to erase the screen and start with a cleared page on the screen in a VT/100 terminal.

## A.3 Procedures Common to Both The Initialization and Simulation Modules

### A.3.1 : random

The function of this procedure is to generate an integer random number, whose value lies between a given maximum and minimum numbers. This procedure may be used by the Initialization module if the queues are chosen to be initialized randomly, and in the

Simulation module if a certain node passes the test and another test is to be scheduled at some random time in the future.

### A.3.2 : enqueue

This procedure is also used by both the Initialization and the Simulation modules. Its main function is to put a message onto a certain queue. It calls some other procedures to perform some specific tasks. These procedures are :

(a) purge

Each time the number of messages in any queue reaches a specific value, this procedure is called in order to delete the messages, that have been assigned as completed. The completed message, is the one with both its Execution and Completion flags having the value COMP (see section 4.2.3). The continuous execution of this procedure prevents the risk of queues overflow.

(b) swap

Every time a message is to be enqueued and the relevant queue is not empty, this procedure is called. Its function is to arrange the messages on the queue according to their time stamps. This sequence is required because it will be followed by the simulator for the execution of messages.

(c) counters

Two message counters are used in this procedure, one is to count the number of all enqueued messages regardless of their type, while the other is only for diagnosis messages.

The operational sequence of procedure 'enqueue' is shown in Figure A.3.

### A.3.3 : dequeue

This procedure is used by the Simulation module. Its function is to take a message out of a queue mainly to be executed.

### A.3.4 : sh-queue

This procedure is used to write into an output file a snapshot of what messages are contained in the queues as well as the simulated times (clocks) of all nodes. The procedure is called after the initialization of the queues in the Initialization module and after every execution of a message in the Simulation module, if the user wish to have all generated

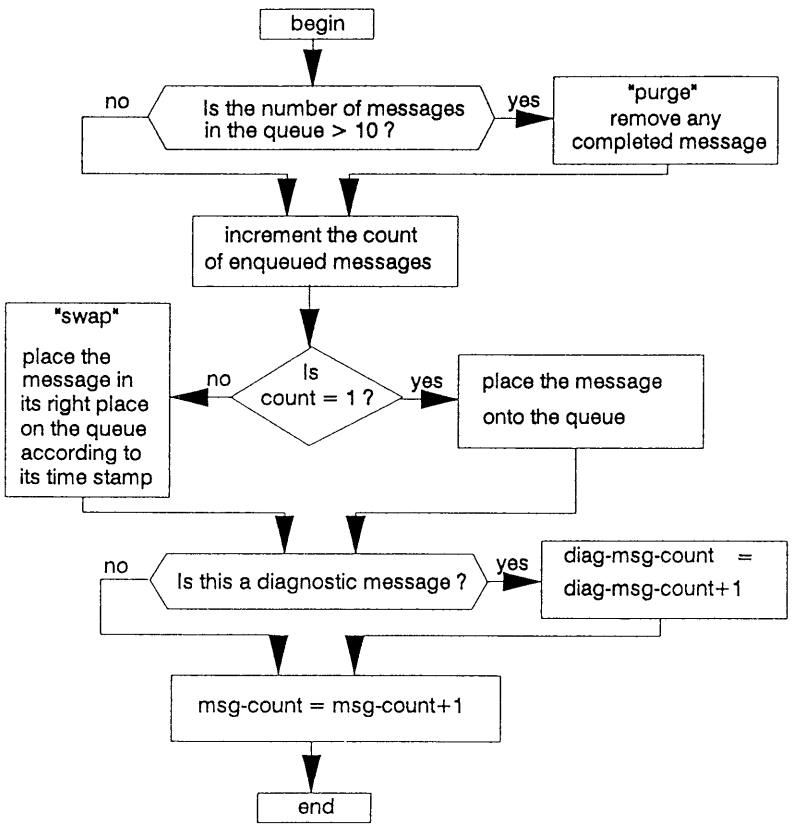



Figure A.3 : Operational sequence of procedure "enqueue"

messages to be written into the file.

## A.4 The Initialization Module

The structure chart of this module is shown in Figure A.4. It calls the following procedures:

### A.4.1 : randinit

This procedure can be chosen by the user as one of two alternatives to initialize the queues. When executed, it will result in generating a random number of test messages (messages of type TEST), and a random number of messages of type INFO. Both types of messages will have random time stamps and placed onto the queues randomly. It is quite possible, however, that the resulting initialization pattern will not be useful for simulating some given fault situations. As an alternative, any initialization pattern of messages can be provided when procedure 'userinit' is used.

### A.4.2 : userinit, ex-msg-file, cr-msg-file

Selecting procedure 'userinit' for initializing the queues will offer the user a choice between reading the initialization pattern from a file, where procedure 'ex-msg-file' will be called or by creating a new file or altering an old file, where procedure 'cr-msg-file' is called. In both cases, the system can be initialized with the messages, that are suitable for the simulation of the required fault situation.

### A.4.3 : init-clock, time-to-stop, time-delays, output-choice

The functions of these procedures were described in sections 4.5.2 - 4.5.5.

## A.5 The Simulation Module

It was mentioned in section 4.3 that this module is the most important and complex in the software, where both the simulation and diagnosis algorithms are employed. The structure chart and the operational sequence of this module are shown in Figures A.5 and A.6 respectively. There are number of procedures, that are common to many sub-modules in the Simulation module and they are described first.

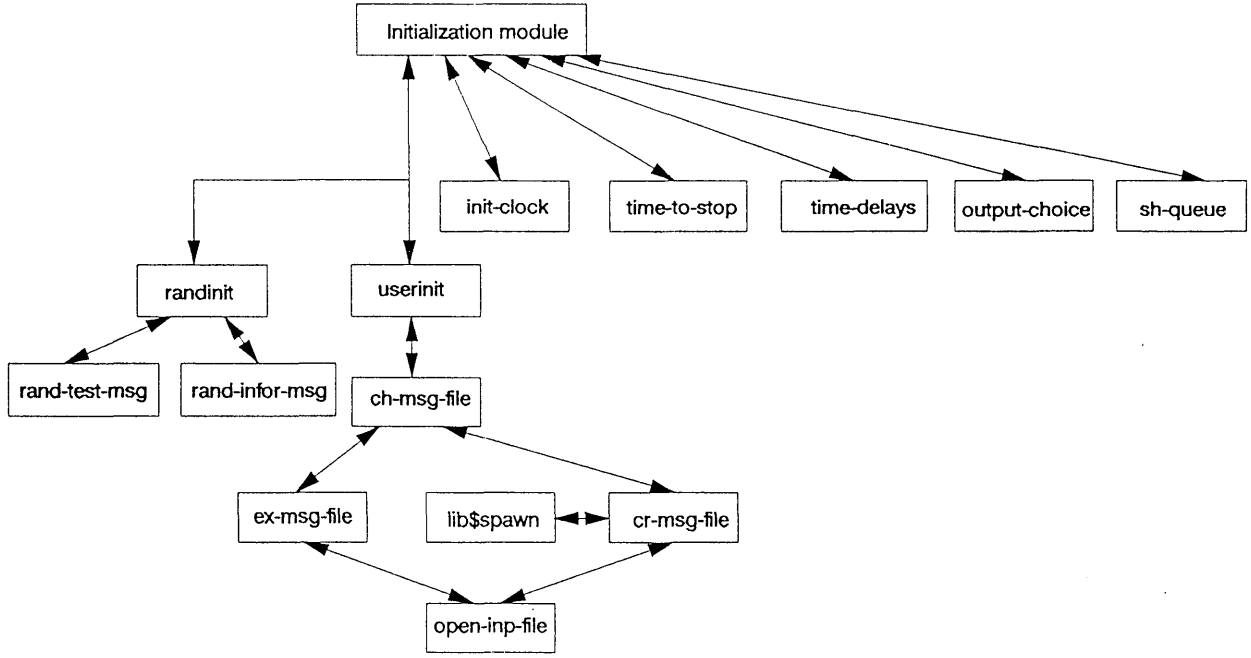



Figure A.4 : Structure chart of the Initialization module

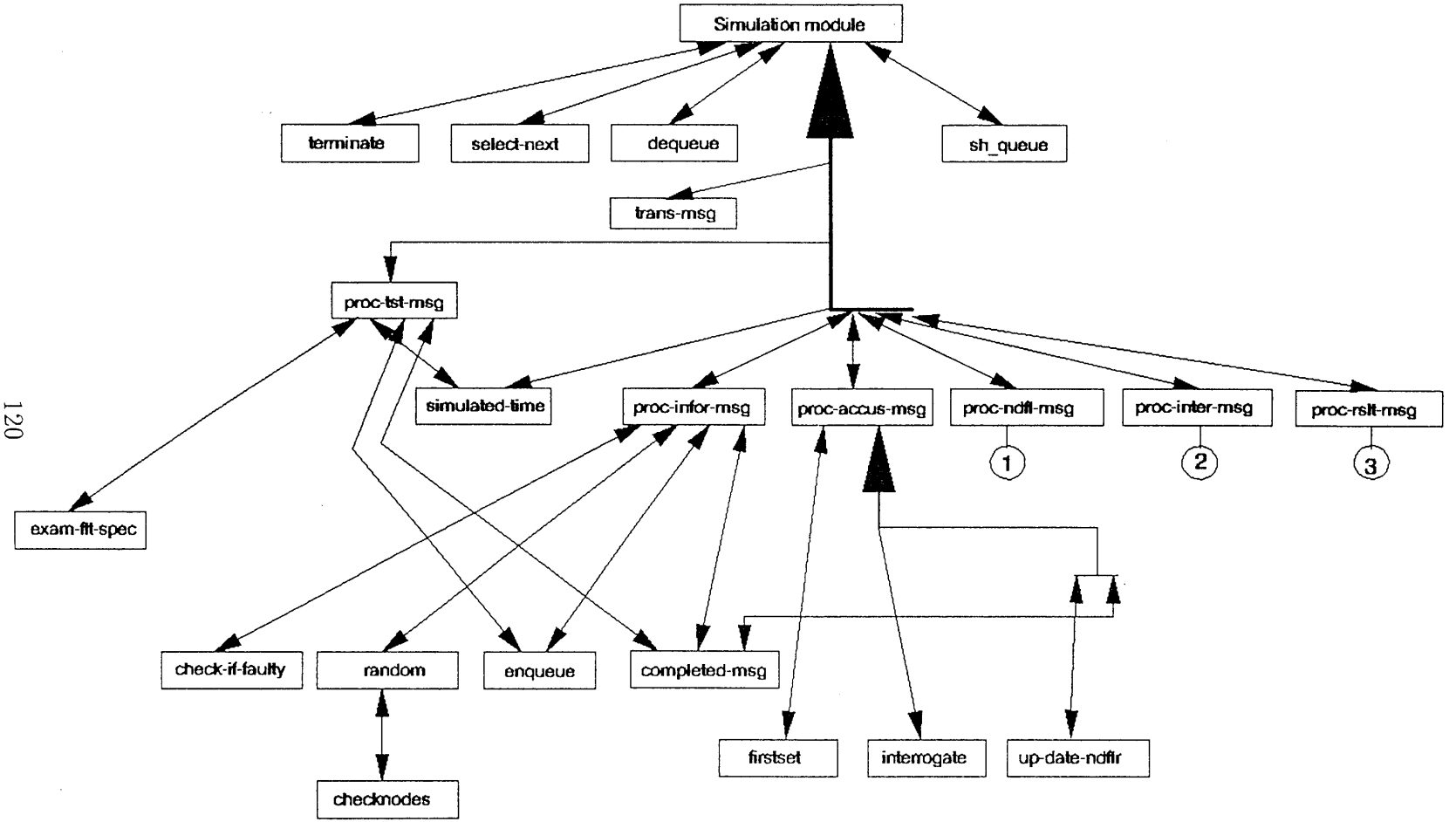



Figure A.5 : Structure chart of the Simulation module

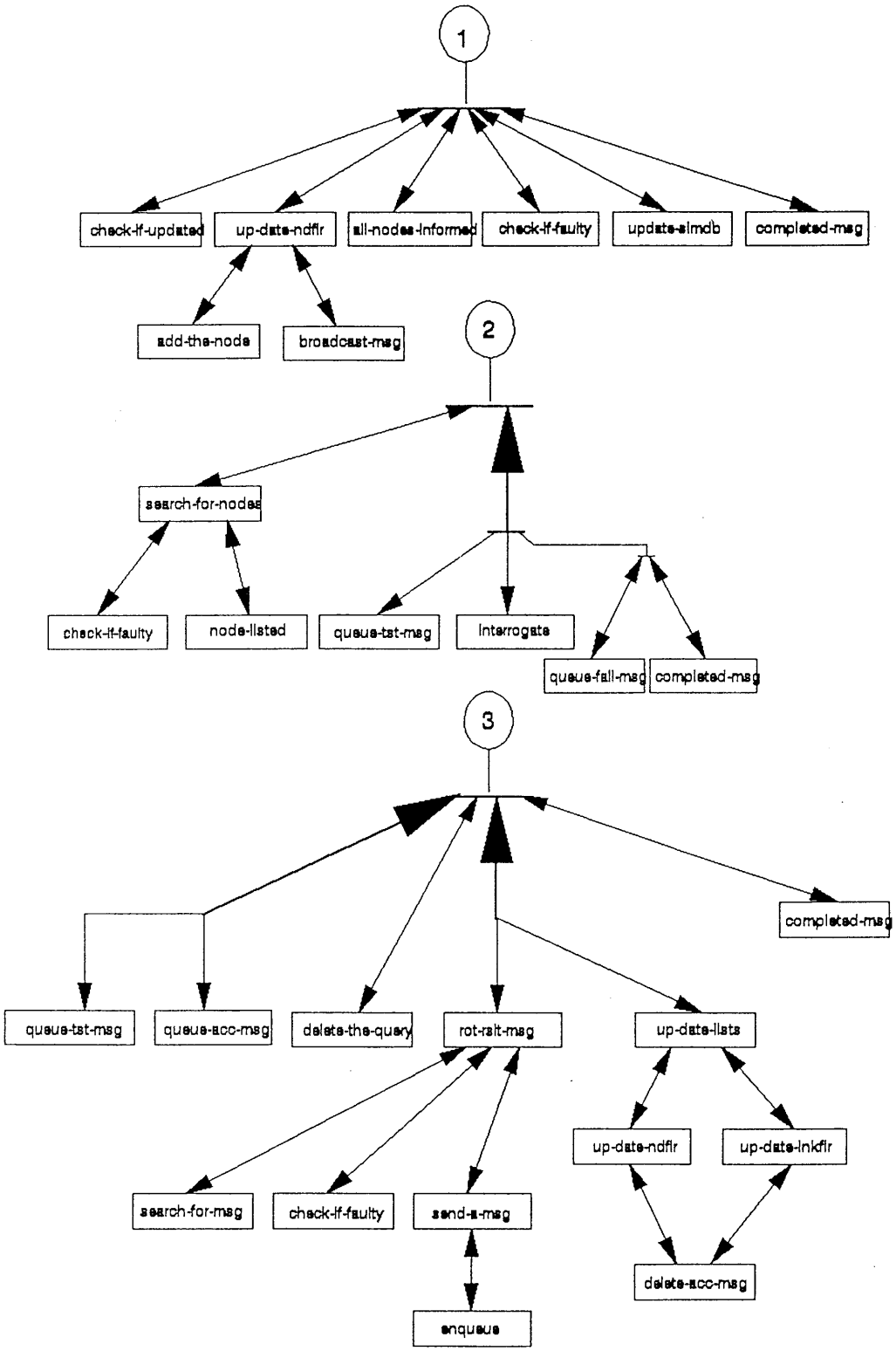



Figure A.5 : cont.

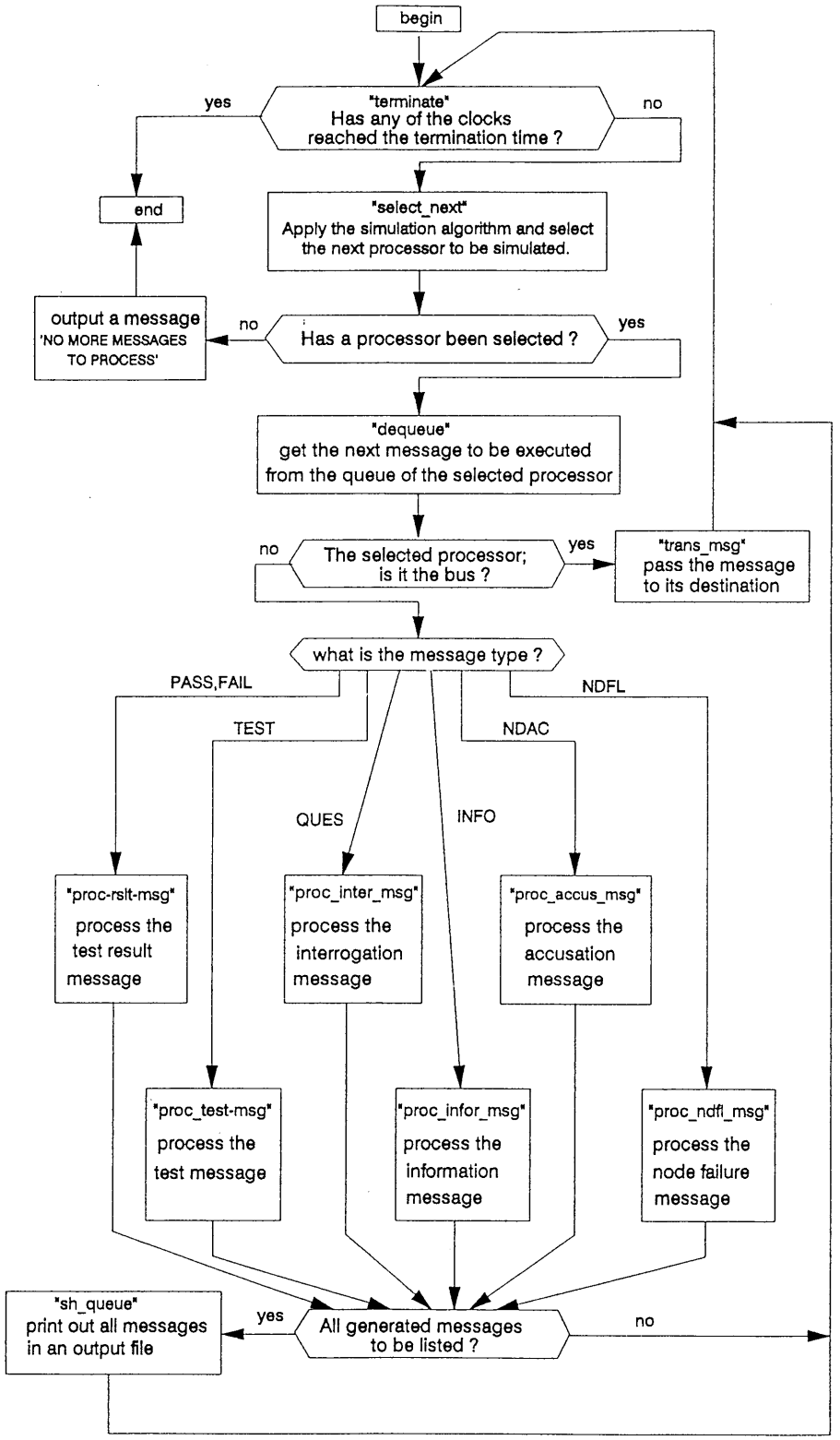



Figure A.6 : Operational sequence of the Simulation module

### A.5.1 : Procedures Common to Many Sub-modules

#### A.5.1.1 : terminate

This procedure is called by the Simulation module each time before a node is selected for simulation. Its function is to compare the simulated times(clocks) of all nodes with a specified termination time. Whenever a clock reaches this value, the program will terminate.

#### A.5.1.2 : check-if-faulty

This procedure is called by many other procedures. Its function is to check through the list of faulty nodes or the list of faulty links, of a specific node in order to make sure that only fault-free elements are included in a specific process.

#### A.5.1.3 : simulated-time

In this procedure, the clock value of the node, that will be simulated next, is updated. The time stamp of the message to be processed by this node and the time required for processing are used in evaluating the updated value of the clock. This is performed as follows :

*if (time stamp* $\leq$ *clock) then*

*clock := clock + processing time*

*else*

*clock := time stamp + processing time*

#### A.5.1.4 : exam-flt-spec

This procedure is used to test the condition of a specific node by examining the fault specification of the system. The construction of these specification and the way they are provided to the simulator were described in section 4.4.2.

#### A.5.1.5 : Interrogate

In this procedure, the node that is being simulated interrogates its fault-free testees about the condition of an accused node. The operational sequence of this procedure is shown in Figure A.7. It calls some other procedures, whose functions are also described in the Figure.

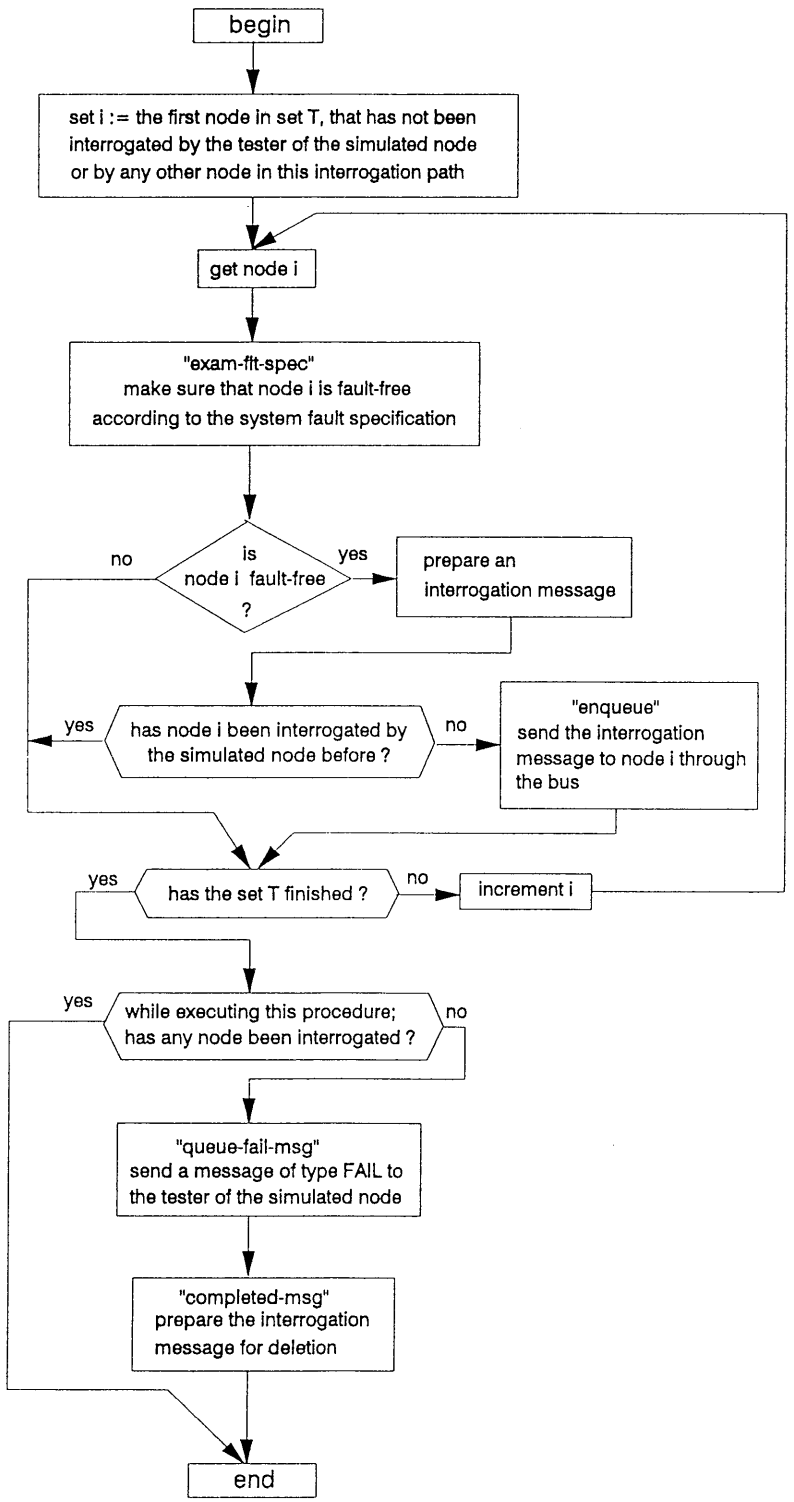



Figure A.7 : Operational sequence of procedure "interrogate"

#### A.5.1.6 : select-next

This procedure employs step(i) of the simulation algorithm described in section 4.2.2. The node, that will be simulated next, is determined by examining the values of the clocks for all the nodes and the earliest time stamp at each queue.

#### A.5.1.7 : up-date-ndflr

In this procedure, the list of faulty nodes, of the node being simulated, is updated, where a node, that has been diagnosed as faulty, is included. Messages of type NDFL are also sent to the fault-free testers of the simulated node, where procedure 'broadcast-msg' is being called, so that they can update their lists of faulty nodes too.

### A.5.2 : Sub-modules of The Simulation Module

#### A.5.2.1 : proc-tst-msg

The function of this procedure is to execute a test message. It has to differentiate between two possible versions of this message. One of these versions is a message, that is related to the testing schedule of the system. If this message is executed and the tested node is found faulty, an accusation message will be generated. In the second version, however, the test is conducted in response to an interrogation message and the test outcome has to be sent back regardless of its type as PASS or FAIL.

#### A.5.2.2 : proc-infor-msg

In this procedure, the simulated node has to update the time stamp of a message of type INFO and pass it to one of its fault-free testees. Whenever a message of this type exists in the system, it will continue circulating throughout its fault-free nodes as long as they are able to communicate and none of their clocks has reached the termination time.

#### A.5.2.3 : proc-inter-msg

The function of this procedure is to execute an interrogation message (message of type QUES). When executing this procedure other procedures are called;

(a) if the node being simulated is a tester of the accused node then procedure 'queue-tst-msg' will be called,

(b) if the node being simulated is not a tester of the accused node, but it has some fault-

free testees, that are not included in the set T, procedure 'interrogate' will be called, and

(c) if the node being simulated neither satisfies (a) nor (b), then procedure 'queue-fail-msg' will be called, which will result in a message of type FAIL to be returned to the source of the interrogation message, which is currently being processed by the node.

**A.5.2.4 : proc-accus-msg**

The function of this procedure is to execute an accusation message. In accordance with the diagnosis algorithm, the simulated node has to find its fault-free testees and then interrogate each one of them. To perform these functions procedures 'firstset' and 'interrogate' are called respectively.

**A.5.2.5 : proc-ndfl-msg**

In this procedure, a message of type NDFL (to broadcast a node failure) is executed. The node being simulated is required to check its list of faulty nodes to make sure that the fault being broadcasted is not already included. Procedure 'check-if-updated' will perform this task. The faulty node may then be added to the appropriate list and the message will be passed to the testers of the simulated node. When all fault-free nodes are informed about the fault (procedure 'all-nodes-informed'), a general data base will be updated (procedure 'up-date-simdb') and the message will be damped to prevent its continuous circulation throughout the system.

**A.5.2.6 : proc-rslt-msg**

The function of this procedure is shown in its operational sequence in Figure A.8.

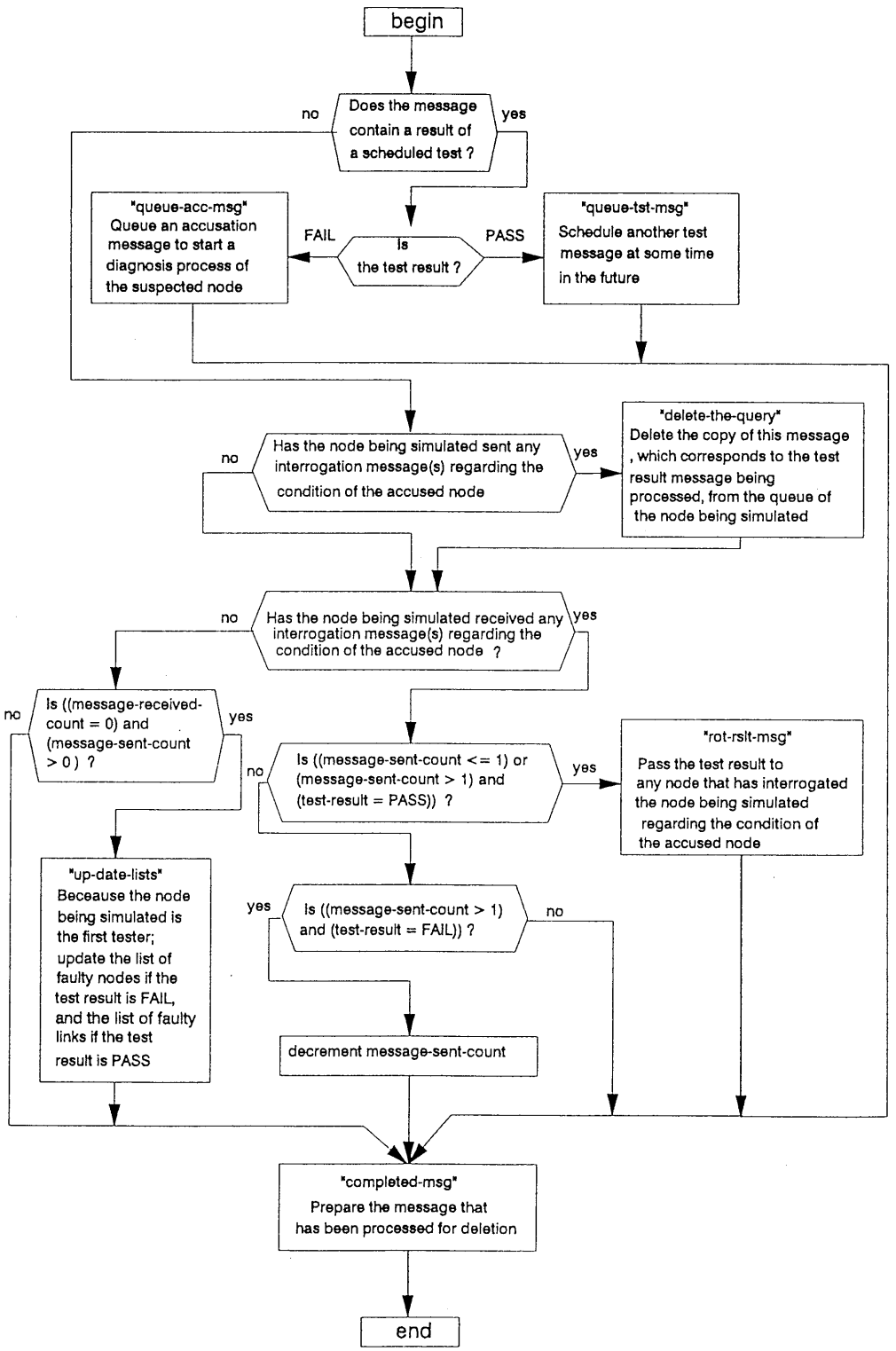



Figure A.8 : Operational sequence of procedure "proc-rslt-msg"

## A.6 The Output Module

The structure chart of the Output module is shown in Figure A.9. It calls the following procedures;

A.6.1 : out-summary

This procedure prints out, both on the screen and into an output file the counts of generated messages and times required for diagnosis.

A.6.2 : out-msg-counts

This procedure prepare statistics about the number of messages enqueued at every queue in terms of their numbers, types, and time stamps.

A.6.2 : out-for-graph

This procedure prints out a table, which contains information about all enqueued messages. The table is divided into a number of groups and each group includes a number of messages. In the group, a message is defined by the address of the node on which it was enqueued and by its time stamp. The data in this table can be represented by a plot, which shows the distribution of messages amongst the queues in time.

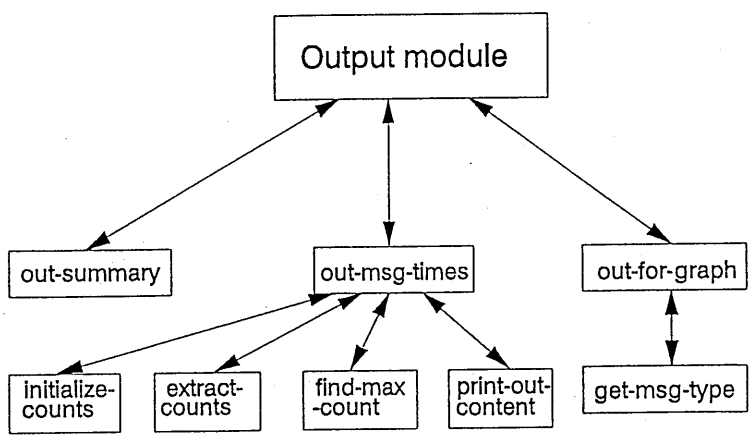



Figure A.9 : Structure chart of the Output module